A VISUAL SIMULATION PLAYGROUND FOR ENGINEERING DYNAMICS

A Thesis

by

DONALD BRIAN FONG

Submitted to the Office of Graduate Studies of
Texas A&M University
in partial fulfillment of the requirements for the degree of

MASTER OF SCIENCE

August 2008

Major Subject: Visualization Sciences

A VISUAL SIMULATION PLAYGROUND FOR ENGINEERING DYNAMICS

A Thesis

by

DONALD BRIAN FONG

Submitted to the Office of Graduate Studies of
Texas A&M University
in partial fulfillment of the requirements for the degree of

MASTER OF SCIENCE

Approved by:

Chair of Committee,	Vinod Srinivasan
Committee Members,	Donald House
	Luciana Barroso
Head of Department,	Tim McLaughlin

August 2008

Major Subject: Visualization Sciences

ABSTRACT

A Visual Simulation Playground for Engineering Dynamics. (August 2008)

Donald Brian Fong, B.S., Northwestern University

Chair of Advisory Committee: Vinod Srinivasan

Past educational studies reveal that students have difficulty making the connection between the mathematical and analytical models used to describe building behavior and the behavior itself. This thesis examines the development and use of visual simulation software as a tool to help students create connections between abstract mathematical models and the real world. A framework for the software was designed and implemented, enabling students to interactively construct, analyze, and evaluate models within a single environment. The software was tested by students in an undergraduate dynamics course to assess its effectiveness as a learning tool. Results are presented through scenarios that demonstrate the extensibility and flexibility of the framework and an analysis of student responses from the Student Assessment of Learning Gains instrument.

TABLE OF CONTENTS

LIST OF FIGURES

FIGURE Page

LIST OF TABLES

CHAPTER I

INTRODUCTION

The literature on student misconceptions of dynamics principles is quite extensive [1]. Misconceptions are very enduring and cannot be easily debunked by standard instruction with lectures, textbooks, demonstrations or laboratories [2]. Students have difficulties developing mathematical models and connecting the response of these models to real system behavior [3]. As a result, students can often complete mathematical calculations correctly without having any idea how their results relate to the performance of a real building. Instead of being a learning exercise, the process begins and ends as a mathematical exercise. Visualization can be used as a tool to create connections between the physical world and more abstract physical and mathematical models.

Since computers have become readily available, it has been widely accepted that computer aided instruction can help students gain a better understand of the subject matter if implemented appropriately [4]. This is especially true for subjects that involve moving objects, three-dimensional structures, or other significant visual components that are not easily represented on a traditional black board. For example, engineering dynamics is the study of motion, but traditional teaching tools, including mathematical models, do not effectively this motion [5].

This thesis focuses on the development and use of visual simulation software to help improve student learning in dynamics. A software framework containing a set of “building blocks” for dynamics models was designed and implemented. The software allows students to construct models of dynamics systems, tweak parameters,

---

The journal model is *IEEE Transactions on Automatic Control.*

and simulate and analyze their behavior.

The design of the framework is illustrated to show how it achieves modularity and facilitates extensibility. In addition, issues encountered when students tested the software are discussed, and improvements made to the software based on this feedback are demonstrated. Lastly, the usefulness of integrating visual simulation software into an undergraduate dynamics course is evaluated.

In summary, this thesis has the following goals:

1. Design and implement an extensible object-oriented framework for visual simulation software that supports undergraduate-level dynamics.
2. Enhance the software based on feedback from user testing sessions.
3. Assess the effectiveness of using the software as a tool for improving student learning in dynamics.

CHAPTER II

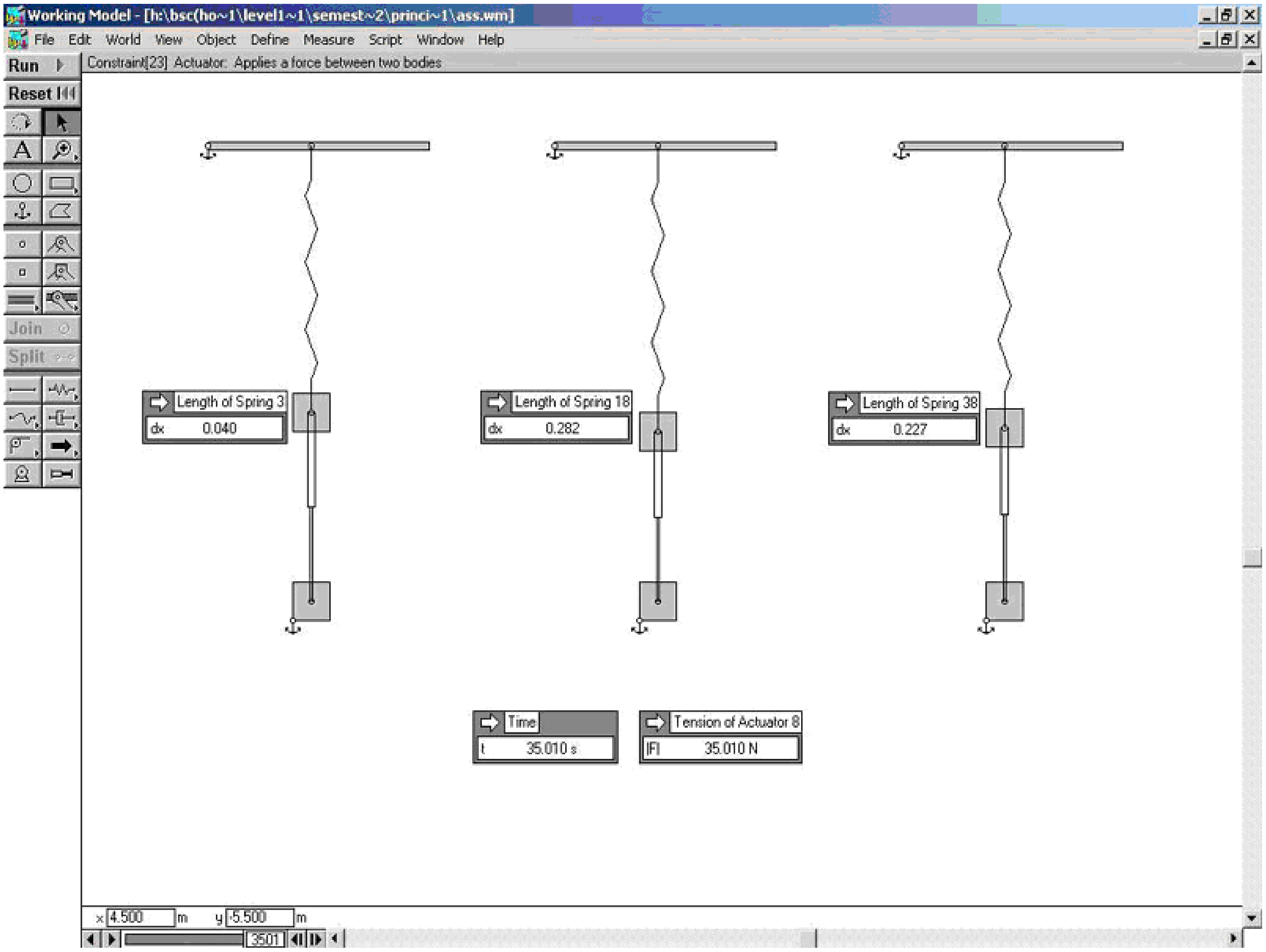
PREVIOUS WORK

Using visualization to help improve student learning is not a new concept. The challenge lies in effectively utilizing visualization in the educational context. Guidelines for effective use and evaluation of visualization were presented by an Association for Computing Machinery working group on “Evaluating the Educational Impact of Visualization,” consisting of members from several universities worldwide [6]. They advocate that visualization for educational purposes (1) be designed for flexibility, (2) capture larger concepts, and (3) map to existing teaching and learning resources. In 1990, an Association of Computing Machinery Special Interest Group on Graphical Display panel consisting of members from academia and industry observed that real-time interactive graphics are remarkably effective at enhancing learning by capturing the dynamic nature of structures when part of a comprehensive teaching strategy [7].

Within the context of dynamics instruction, two previous studies are particularly notable: the use of Working Model at the Georgia Institute of Technology [4] and the development of BEST Dynamics at the University of Missouri-Rolla [5].

Working Model is a program that integrates advanced simulation techniques with an easy-to-use graphical interface as shown in Figure 1. It enables users to design and test prototypes of mechanical and structural systems, using two-dimensional animation to show the motion of the objects involved in the simulation. Although Working Model allows students to design and visualize the working of mechanical systems, it assumes an understanding of the underlying physical law of engineering [4] and has a fairly steep learning curve [5]. The assumption that the user already knows and understands dynamics makes Working Model a suitable design tool that is valuable for tackling “what-if” design problems. However, this does not necessarily make it a suitable teaching tool for students with little or no prior exposure to dynamics.

The objective of the “BEST” (Basic Engineering Software for Teaching) Dynamics project was to improve the teaching and learning of engineering dynamics [5]. The software contains a predefined set of problems ranging from particle systems to rigid body kinematics and was designed to allow student use without supervision. Figure 2 shows an example of one of the predefined problems. Users are allowed to start, stop, and reset a simulation, step through a simulation, and vary inputs to explore various scenarios. Important variables in the simulation like position, velocity, forces, etc. are output as numerical values. For some of the problems, solutions are available, which include diagrams, vector directions, and step-by-step solution procedures. Elementary theory sections are also available for each class of problem. The predefined problems are enhanced versions of textbook problems and prove useful to reflective learners, who also learn well through traditional textbook and lecture format. However, active learners drew few benefits. In addition, the underlying misconceptions could not be identified or addressed using this software, a key element in enhancing the learning of all students.


Working Model - [h:\bsc(ho~1\level1~1\semest~2\princi~1\ass.wm]
File Edit World View Object Define Measure Script Window Help
Run
Reset
Join
Split

Constraint[23] Actuator: Applies a force between two bodies

Length of Spring 3
dx 0.040

Length of Spring 18
dx 0.282

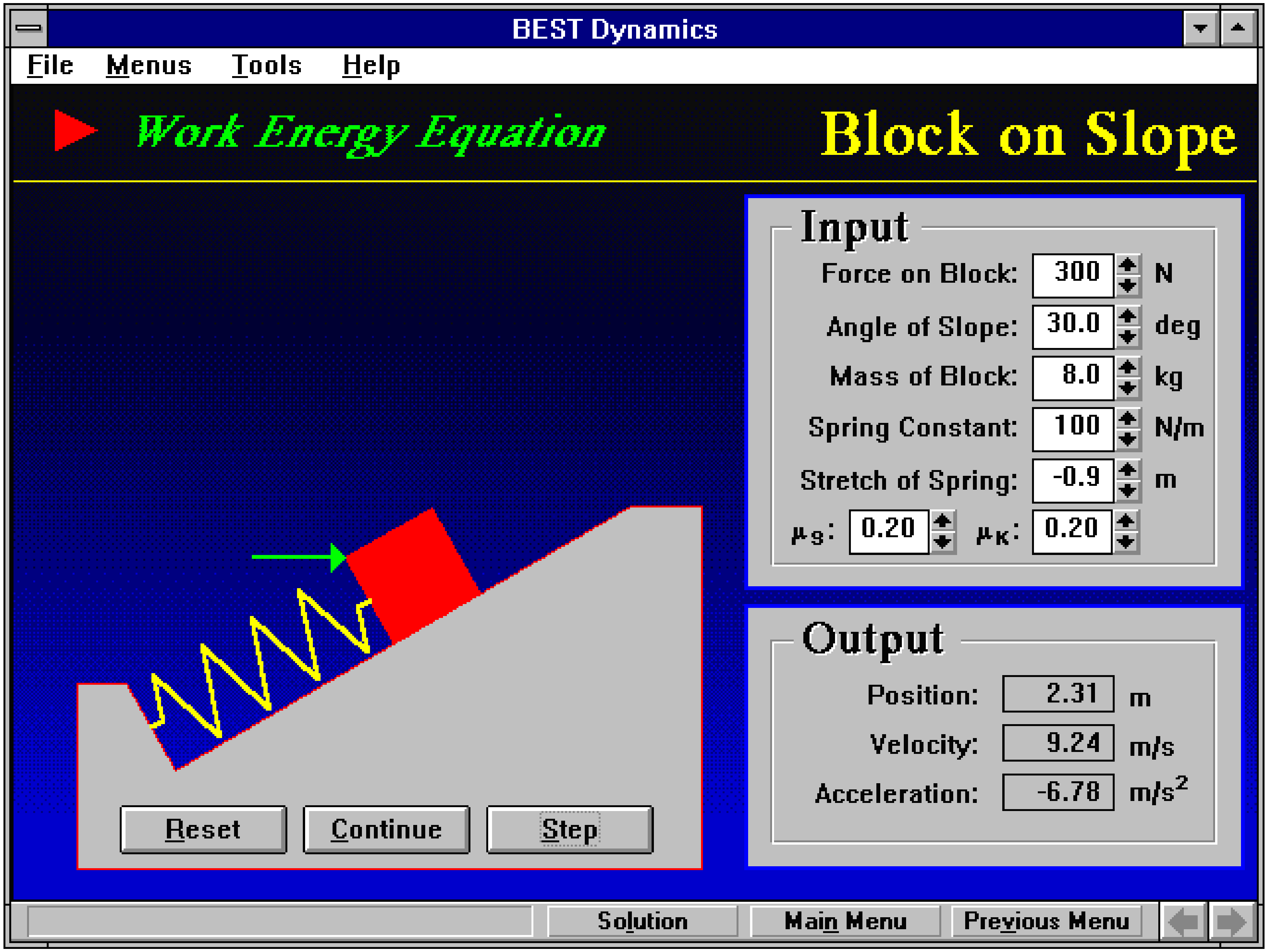
Length of Spring 38
dx 0.227

Time
t 35.010 s

Tension of Actuator 8
|F| 35.010 N

x 4.500 m y -5.500 m
3501

Fig. 1. Working Model graphical interface



Input

Force on Block:	300	N
Angle of Slope:	30.0	deg
Mass of Block:	8.0	kg
Spring Constant:	100	N/m
Stretch of Spring:	-0.9	m

μ_S: 0.20 μ_K: 0.20

Output

Position:	2.31	m
Velocity:	9.24	m/s
Acceleration:	-6.78	m/s^2

Fig. 2. BEST Dynamics graphical interface

CHAPTER III

METHODOLOGY AND IMPLEMENTATION

This chapter focuses on the design methodology behind the framework for the visual simulation software. The framework is designed to be very modular and easily expandable. To make the framework easier to understand, it can be broken down into three major components, or "engines": the physics engine, the user interface engine, and the rendering engine.

The physics engine is in charge of simulating the objects based on the laws of Newtonian physics. The rendering engine is responsible for drawing the objects on the screen. And the user interface engine is in charge of taking user input and interacting with the physics and rendering engines based on the input.

Figure 3 illustrates how the three engines communicate with each other. The physics engine controls the scene objects by updating their positions and orientations while the simulation is running. The user interface engine can communicate with both the physics engine and the rendering engine. For example, if the user wants to add a new object to the scene, the user interface engine relays the message to the physics engine to add a new object. Similarly, if the user wants to toggle the display of labels, the user interface engine will send a message to the rendering engine to toggle the labels.

Figure 4 shows the object hierarchy and object relationships to the interfaces. Almost every class inherits from a base Entity class as well as a number of interfaces. An interface is a collection of method declarations without implementations. When a class implements an interface, it promises to implement all of the methods declared in that interface. The interfaces enforce essential functionality, which allow the objects to be used in the physics, rendering, and user interface engines. The object hierarchy

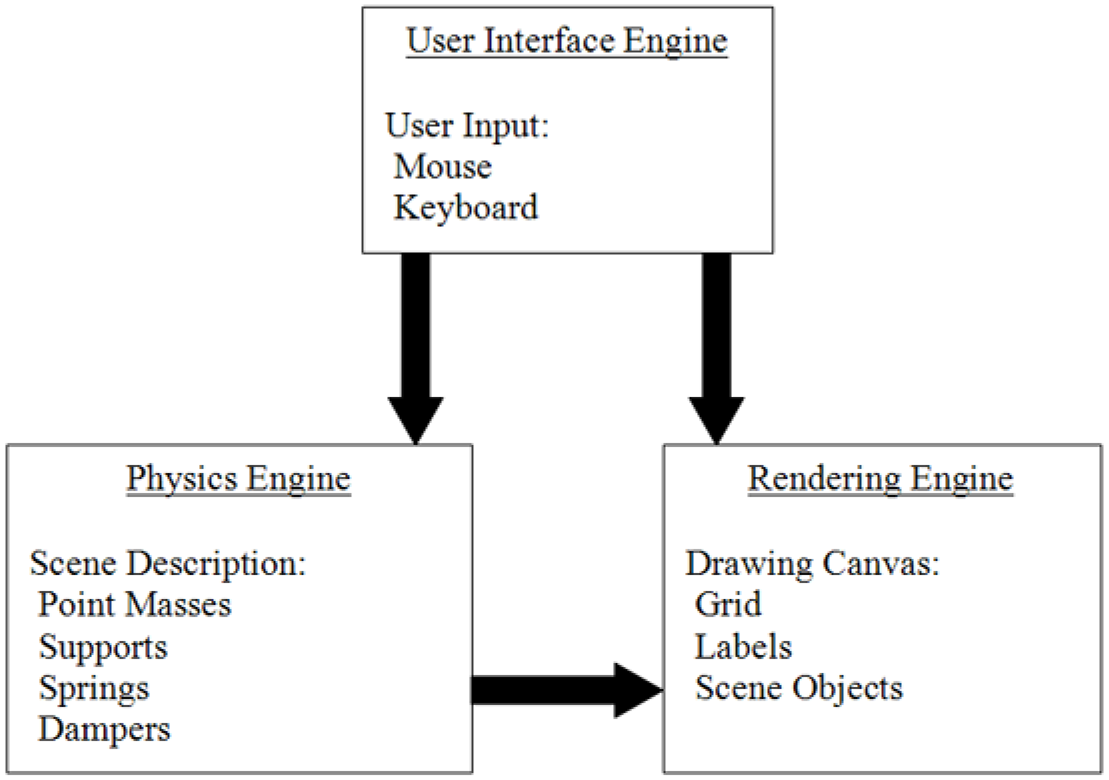


Fig. 3. Engine relationships diagram

helps to organize the objects into logical groupings and to take advantage of the benefits of inheritance. The following sections will examine the three engines and explain how the classes and interfaces are used within each engine.

A. The Physics Engine

The physics engine is the heart of the visual simulation software framework. Its purpose is to simulate the motion of the objects in the scene based on Newton's laws of motion. The physics engine updates the positions and orientations of objects in the scene and notifies the rendering engine, which then updates the display with the new states of the objects.

1. Building Blocks

The framework contains a set of "building blocks", or objects, that can be connected in various combinations to create dynamics models normally encountered in an undergraduate dynamics course. These objects implement interfaces which enable them to be used within the three engines. Within the framework, the "building blocks" are grouped into three main classes: `ConnectableObject`, `Force`, and `Connector`.

a. ConnectableObject

A `ConnectableObject` is an object that can have forces applied to it. It is called a connectable object because objects that apply forces can be connected to it. For example, a spring can be attached to a connectable object. Each connectable object has a force accumulator which is used to store the sum of all of the forces acting on the object during a simulation step.

Figure 5 shows the `ConnectableObject` class hierarchy. The two "building

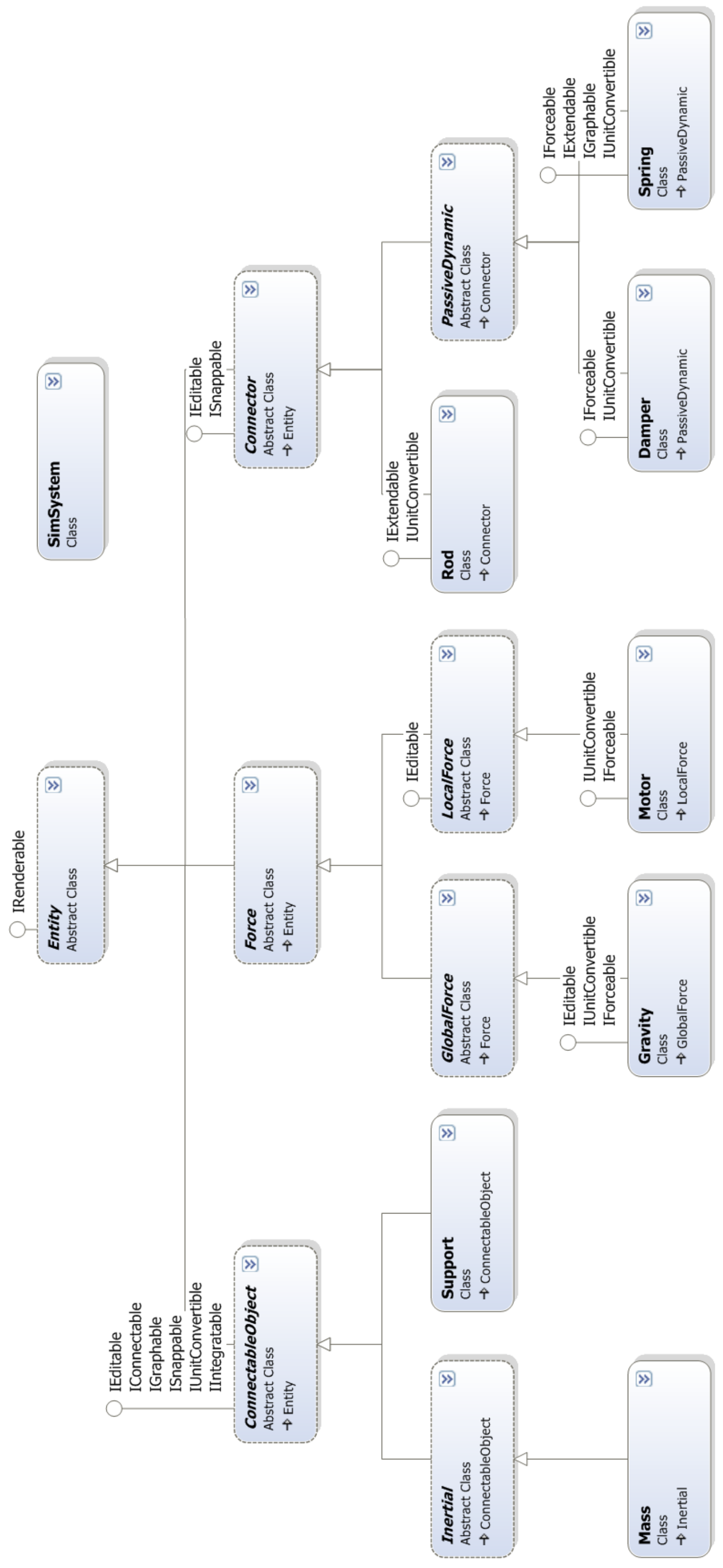


Fig. 4. Object hierarchy and interface diagram

blocks" that derive from `ConnectableObject` are `Mass` and `Support`. The key difference between a mass and a support is that a support does not react to forces acting upon it. A support remains fixed even if forces are acting on it.

b. Force

Figure 6 illustrates the `Force` class hierarchy. Objects that derive from the `Force` class apply a single force to objects that implement the `IIntegratable` interface (described in more detail in the next section). However, the number of objects that the force affects may vary. In some cases, the force may only affect one object, but in other cases, it may affect every object in the scene. To differentiate between these two types of forces, we have two additional abstract classes, `GlobalForce` and `LocalForce`.

A `GlobalForce` object will apply a force to every object in a scene. Gravity and wind are examples of global forces. A `LocalForce` object will only apply a force to the object that it is connected to. If we attach a motor to a mass, for example, the force generated by the motor will only affect the mass it is attached to. It will not affect other masses in the scene.

c. Connector

The `Connector` class contains objects that generate forces but must be connected to two connectable objects to function properly. This includes springs, dampers, and rods. Figure 7 shows these objects in the `Connector` class hierarchy.

2. Interfaces

The easiest way to understand what functionality an object has is by examining the interfaces that it can implement. The interfaces that are important to the physics

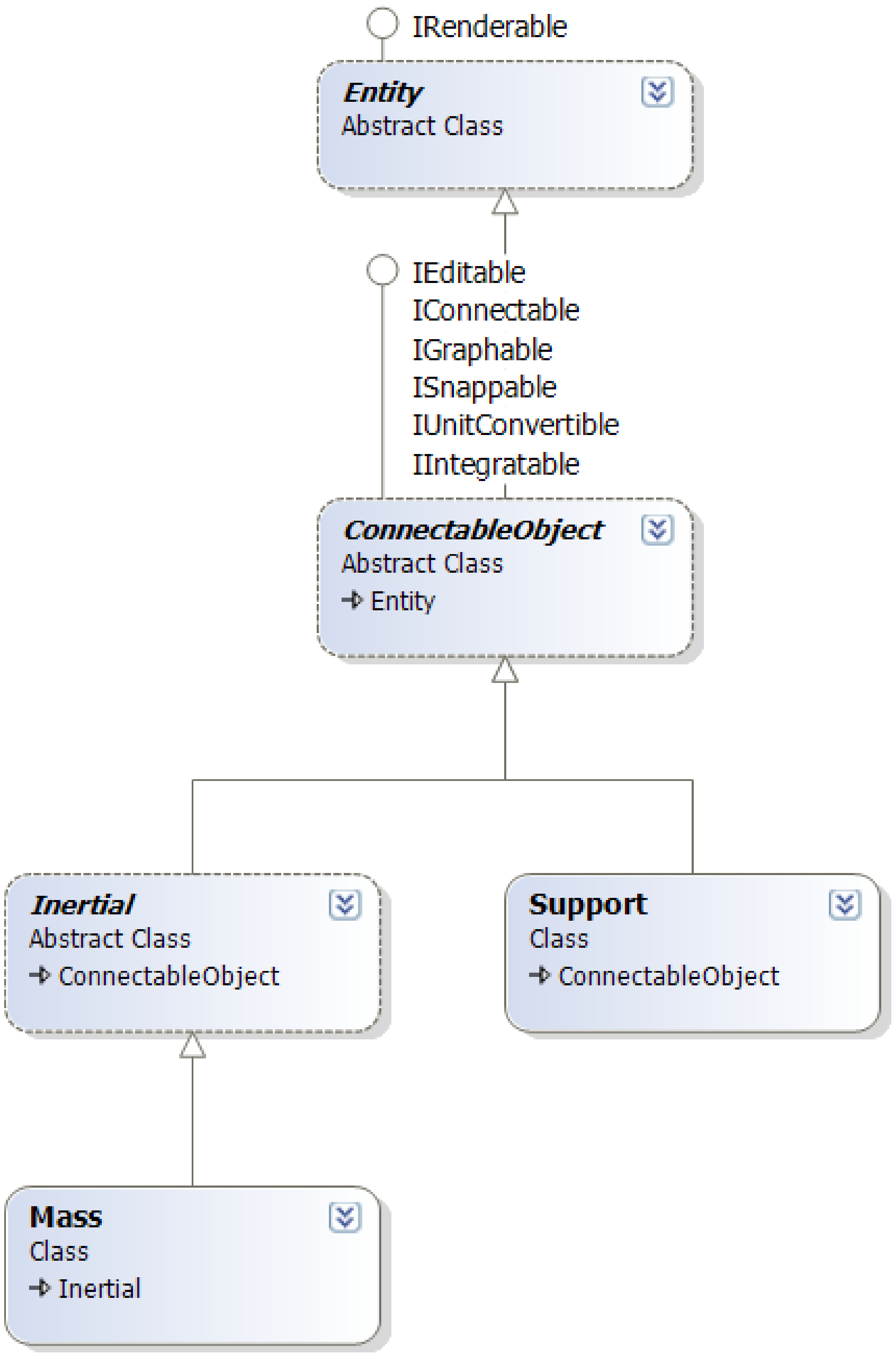


Fig. 5. ConnectableObject hierarchy

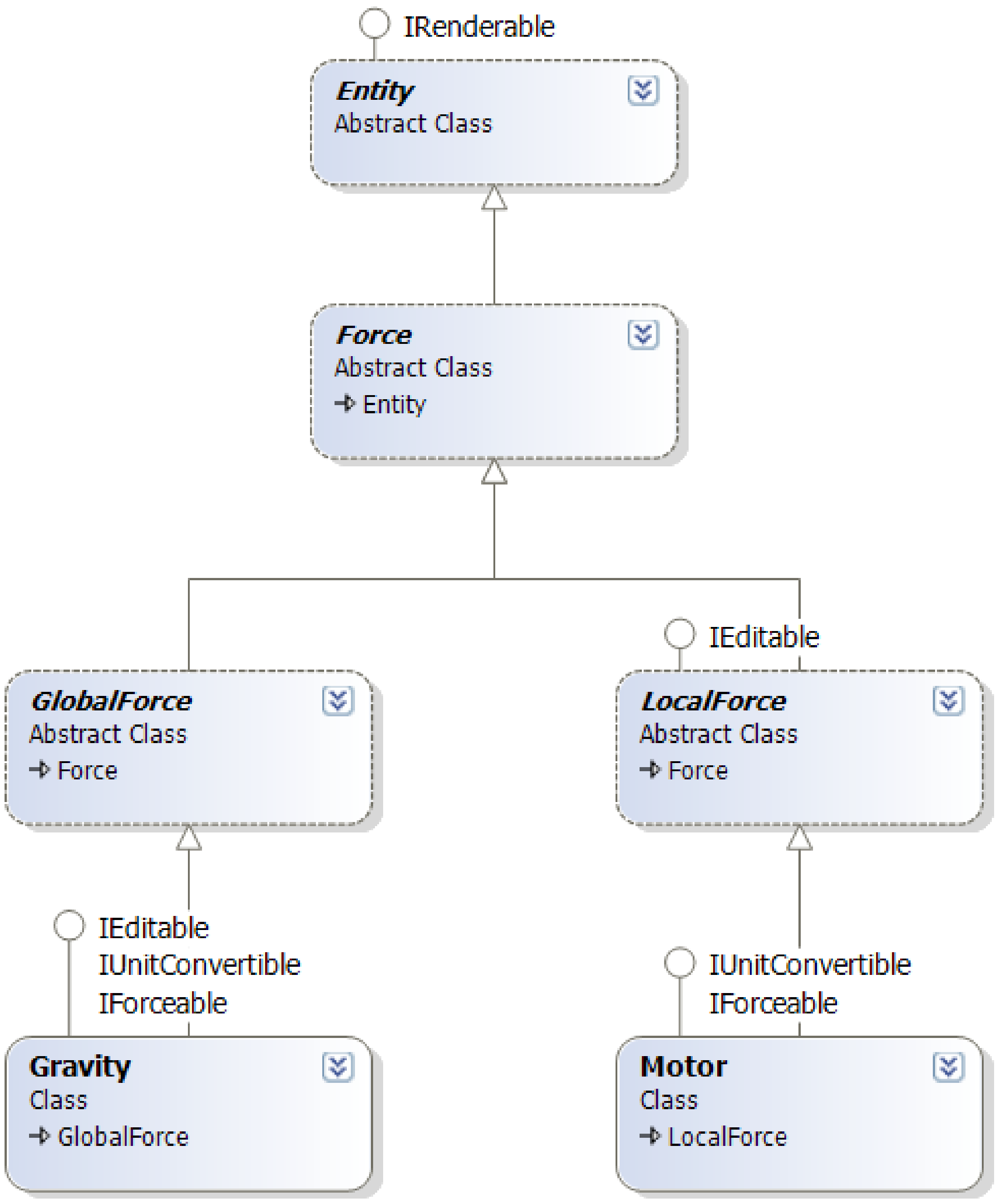


Fig. 6. Force hierarchy

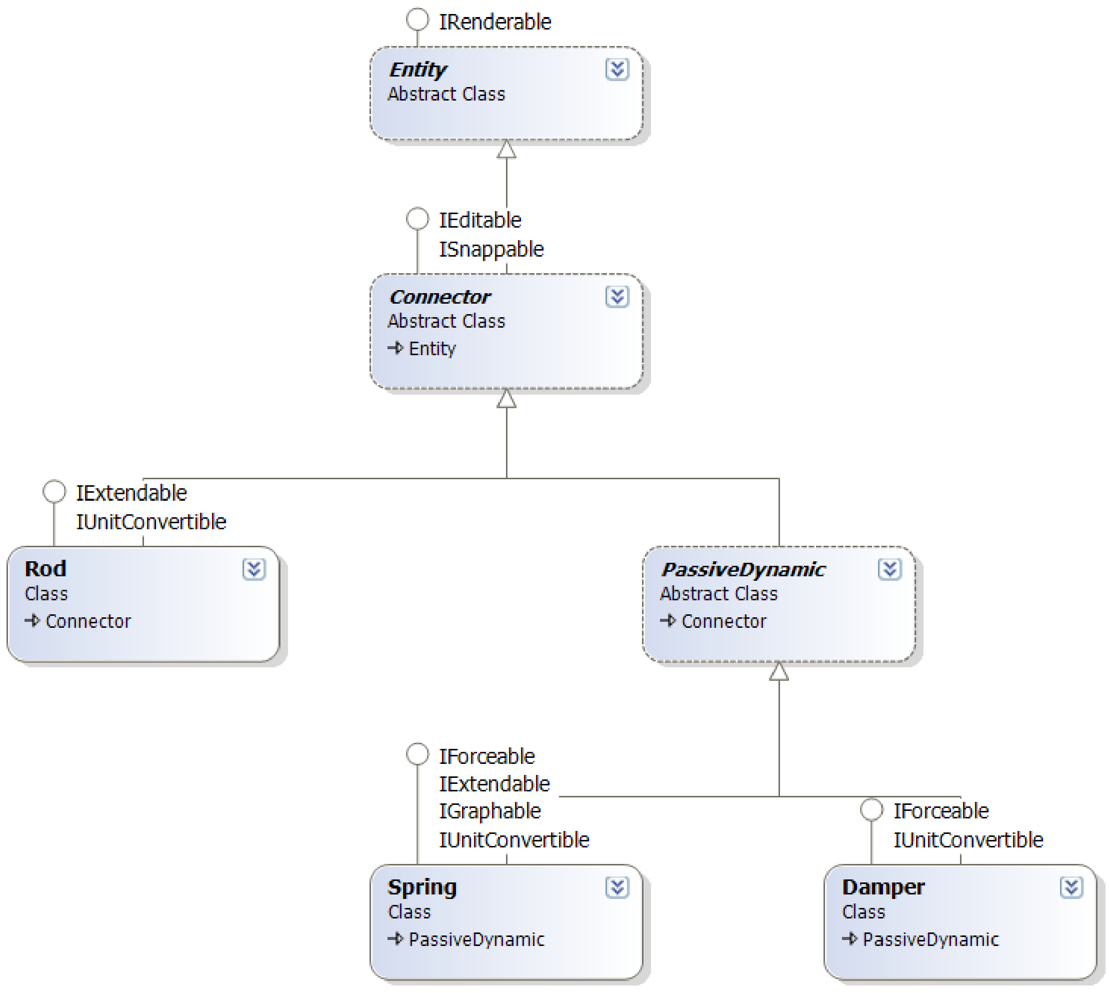


Fig. 7. Connector hierarchy

engine are `IIntegratable` and `IForceable`. The following sections outline the functionality of these interfaces.

a. IIntegratable

Objects that implement the `IIntegratable` interface can be put into the physics engine to be simulated. Three important methods must be defined by an object that implements the `IIntegratable` interface. These methods ensure that every "integratable" object defines how to update its state, how to set its initial conditions, and how to reset itself to initial conditions. The code for the `IIntegratable` interface is shown below.

```
public interface IIntegratable
{
    void UpdateState(Vector2d pos, Vector2d vel);
    void SetInitCondition();
    void Reset();
}
```

b. IForceable

Objects that implement the `IForceable` interface can be put into the physics engine to apply forces to "integratable" objects. An object that implements the `IForceable` interface must define an `ApplyForce()` function that calculates a force and adds it the appropriate force accumulator of the appropriate object or objects. The interface definition for `IForceable` is shown below.

```
public interface IForceable
{
    void ApplyForce(State objects, List<ConnectableObject>
         objList, double t);
}
```

How the applied force is calculated depends completely on the object that implements the interface. For example, a spring will apply a force proportional to the distance between the two objects it is connected to, whereas a damper will apply a force proportional to the relative velocity between the two objects it is connected to.

3. The Simulation System

Each time a timestep is taken, the `SimSystem` class is in charge of updating the objects' states based on the current state of the system. The simulation system does this by using objects that implement the `IIntegratable` and `IForceable` interfaces previously mentioned The two most important functions of the `SimSystem` class are `SystemDynamics()` and `RK4()`.

To calculate the correct accelerations for each "integratable" object, the simulation system must determine all of the forces acting on the objects. This is done by the `SystemDynamics()` function which loops through every `IForceable` object and calls its `ApplyForce()` method. Since each `IForceable` object knows which object, or objects, to apply forces to, each "integratable" object has accumulated all of the forces acting on it by the end of the loop. Using the accumulated forces, accelerations are computed for the "integratable" objects.

After the accelerations are calculated, `RK4()` uses fourth-order Runge-Kutta integration [8] to compute the next state for each object based on the timestep. After the simulation system calculates the new state of each object, `UpdateState()` is called on each `IIntegratable` object to ensure that all of its important information is updated to the latest state. For example, this method updates the position and velocity for a point mass. Once every "integratable" object has been updated to its new state, the physics engine notifies the rendering engine, which updates the display.

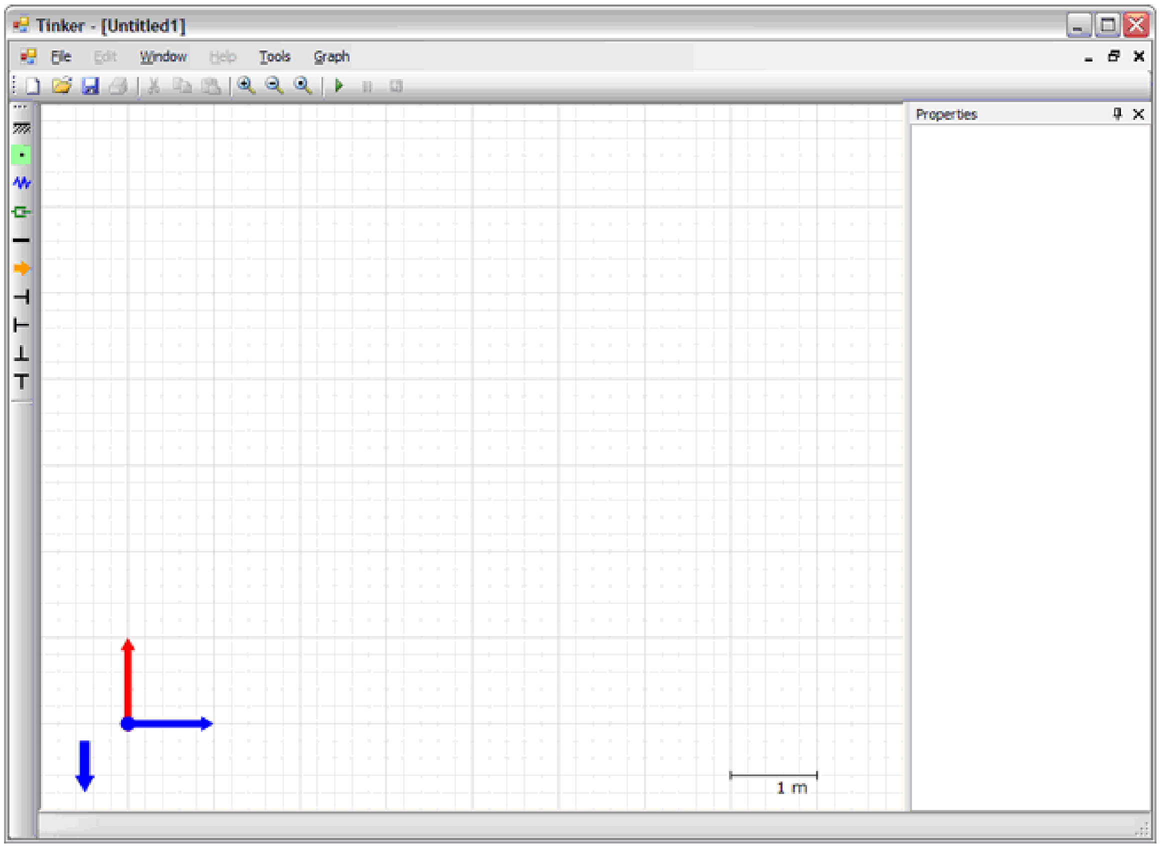


Fig. 8. User interface

B. The User Interface Engine

The user interface is how the user communicates with the software and sees the results of the simulation. The framework uses Windows Forms, which is part of Microsoft's .NET Framework, for its user interface.

Figure 8 shows a screenshot of the user interface. Because Windows Forms uses native Microsoft Windows interface elements, the application looks and behaves like a regular Windows application.

1. Interfaces

Similar to the physics engine, the easiest way to determine the functionality of an object in the user interface engine is by examining the interfaces that it can implement. The interfaces that the user interface engine uses are `IEditable`, `IUnitConvertible`, and `IGraphable`. The following sections will examine these interfaces and how they are used by the user interface engine.

a. IEditable

The `IEditable` interface is used by the user interface engine to handle mouse and keyboard input from the user. Every object that can be edited by the user implements the `IEditable` interface, which ensures that the object defines the following three functions: `Hit()`, `Move()`, and `MoveAbsolute()`. The definition of the `IEditable` interface is shown below.

```
public interface IEditable
{
    bool Hit(Vector2d p, Converter conv);
    void Move(Vector2d offset);
    void MoveAbsolute(Vector2d p);
}
```

The `Hit()` method is used by user interface engine to determine whether an object has been selected. Every time the user clicks a mouse button, the user interface engine loops through all of the `IEditable` objects and calls their `Hit()` method to determine whether or not the object has been selected. The `Hit()` function is also used when a user is trying to connect one object to another object. While the user drags the connection point around, the user interface engine runs hit tests to determine whether or not the point is hitting an object. If the connection point is hitting an object when the user releases the mouse, then user interface engine will send this information to the physics engine so that it can connect the two objects in the simulation.

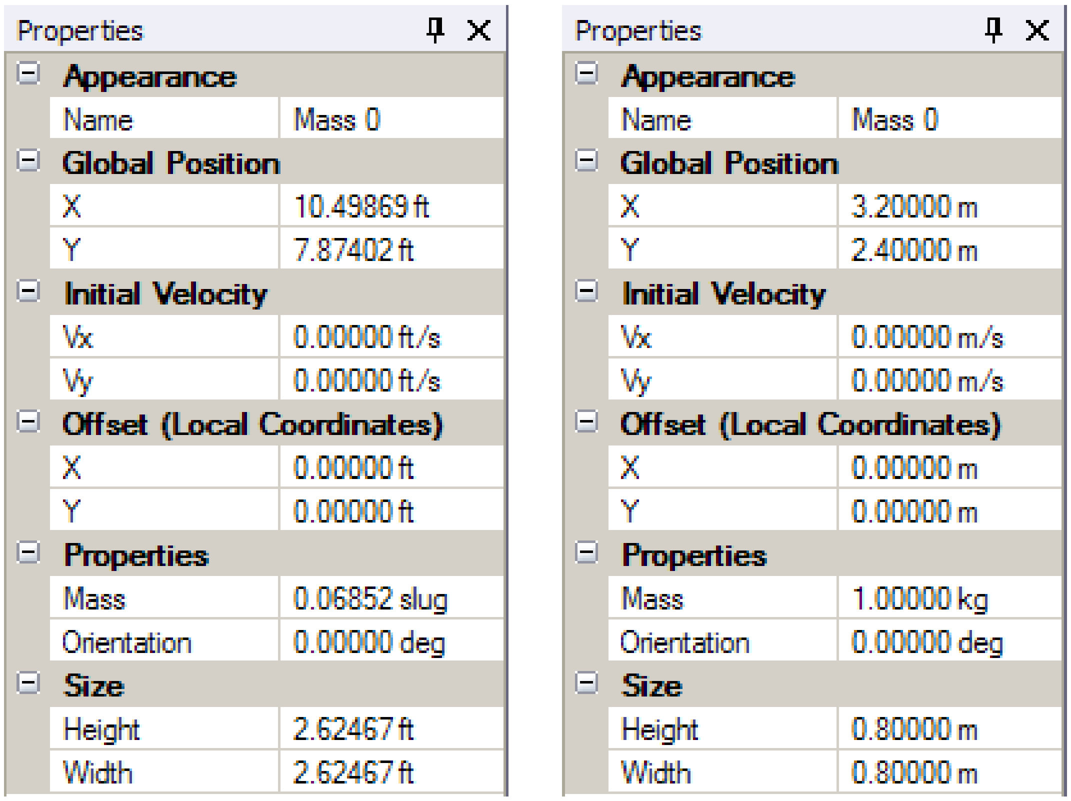
When the user clicks on an object and drags it around the screen, the user interface engine calls the `Move()` method of the object. This function defines how to move an object by a specified offset from its current position. The `MoveAbsolute()` method is similar except that it moves an object to an absolute position instead of a relative position. This function is useful for user interface features like snapping to the grid.

b. IUnitConvertible

An object that implements the `IUnitConvertible` interface can have its properties displayed in different unit systems. This interface guarantees that the object defines get and set methods for a `UnitSystem` property. When the user changes the unit systems in the user interface, the user interface engine loops through all of the `IUnitConvertible` objects and sets each one to the unit system that the user selected. The code for the `IUnitConvertible` interface is displayed below.

```
public interface IUnitConvertible
{
    UnitSystem UnitSystem { get; set; }
}
```

Internally, the values for each object are stored in SI units. However, when the `UnitSystem` property is changed, the user interface engine will display the values in the new unit system by converting the stored values. Figure 9 displays the properties of a mass displayed in SI units and English units.



Properties

Appearance	
Name	Mass 0
Global Position	
X	10.49869 ft
Y	7.87402 ft
Initial Velocity	
Vx	0.00000 ft/s
Vy	0.00000 ft/s
Offset (Local Coordinates)	
X	0.00000 ft
Y	0.00000 ft
Properties	
Mass	0.06852 slug
Orientation	0.00000 deg
Size	
Height	2.62467 ft
Width	2.62467 ft

Properties

Appearance	
Name	Mass 0
Global Position	
X	3.20000 m
Y	2.40000 m
Initial Velocity	
Vx	0.00000 m/s
Vy	0.00000 m/s
Offset (Local Coordinates)	
X	0.00000 m
Y	0.00000 m
Properties	
Mass	1.00000 kg
Orientation	0.00000 deg
Size	
Height	0.80000 m
Width	0.80000 m

Fig. 9. Properties in different unit systems

c. IGraphable

The user interface engine uses the `IGraphable` interface to allow the user to add, delete, and save graphs. An object that implements the `IGraphable` interface must maintain a list of graphs associated with it, define how to create a new graph, and define how to initialize graphs from a saved file. The `IGraphable` interface is defined below.

```
public interface IGraphable
{
    List<GraphWrapper> Graphs { get; set; }
    GraphWrapper CreateGraph();
    void InitializeGraphs();
}
```

An `IGraphable` object must define get and set methods for a list of its graphs. While it is possible to store the graphs in a global list in the user interface engine, storing them with each `IGraphable` object minimizes bookkeeping. In effect, this simplifies the process of saving a scene file and reloading the graphs when the scene file is opened again. When the user chooses an object and creates a graph, the user interface engine calls the `CreateGraph()` method of the object. This method creates a new graph associated with the object and brings it to the front of the drawing canvas.

For graphs that have been saved in a scene file, the user interface engine calls the `InitializeGraphs()` function on `IGraphable` objects when the file is loaded. This function ensures that the graphs appear in the same state as they did when the file was saved. This includes properties like the size and location of the graphs on the drawing canvas.

To save development time, a preexisting graphing library called ZedGraph was used to generate graphs. ZedGraph is an open source library written in C# for creating 2D line and bar graphs of arbitrary datasets. Figure 10 shows an example

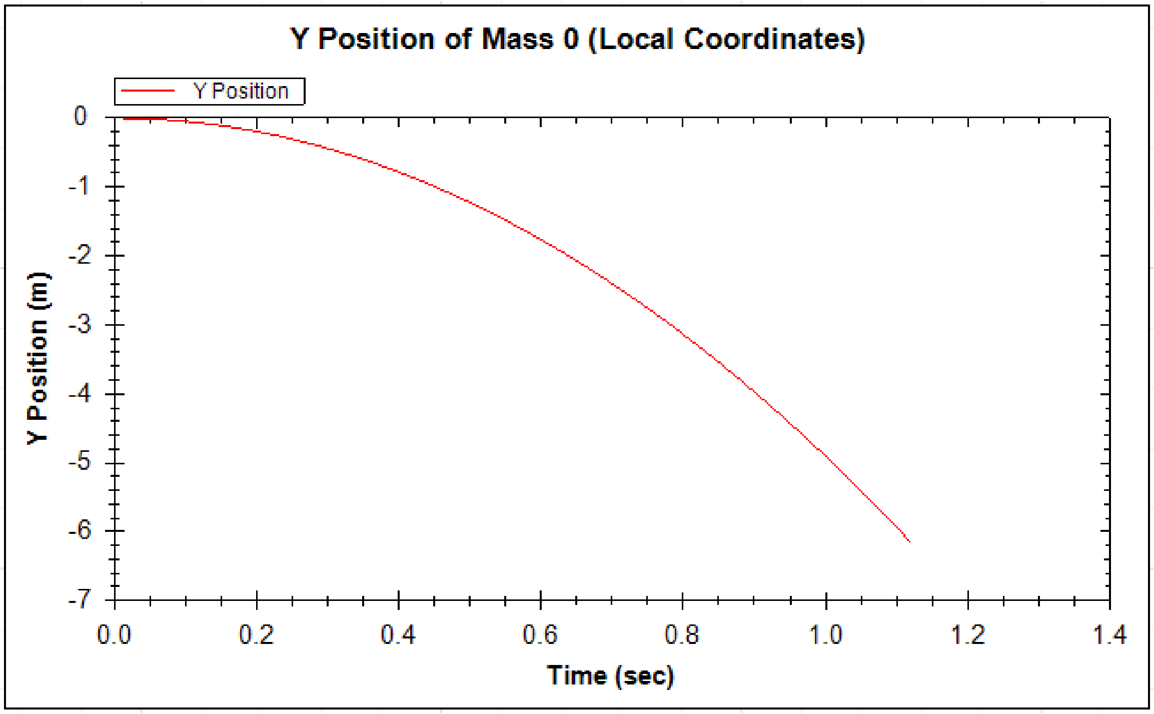


Fig. 10. Example of a graph

of a graph generated in the software using the ZedGraph library.

There are additional features that can be accessed by right-clicking on a graph in the program. Figure 11 shows the menu that appears after you right-click on a graph. Using this menu, the user can perform a variety of additional tasks. “Copy” will allow the user to copy and paste the graph into another program, like a Word document. “Save Image As” allows the user to save the graph as an image file. “Show Point Values” enables the user to move the mouse over the curve in the graph, and the point values will appear as the mouse hovers over a specific point on the graph. “Export Values To File” lets the user saved the simulation data to a file that can be used for more detailed analysis in an external program.

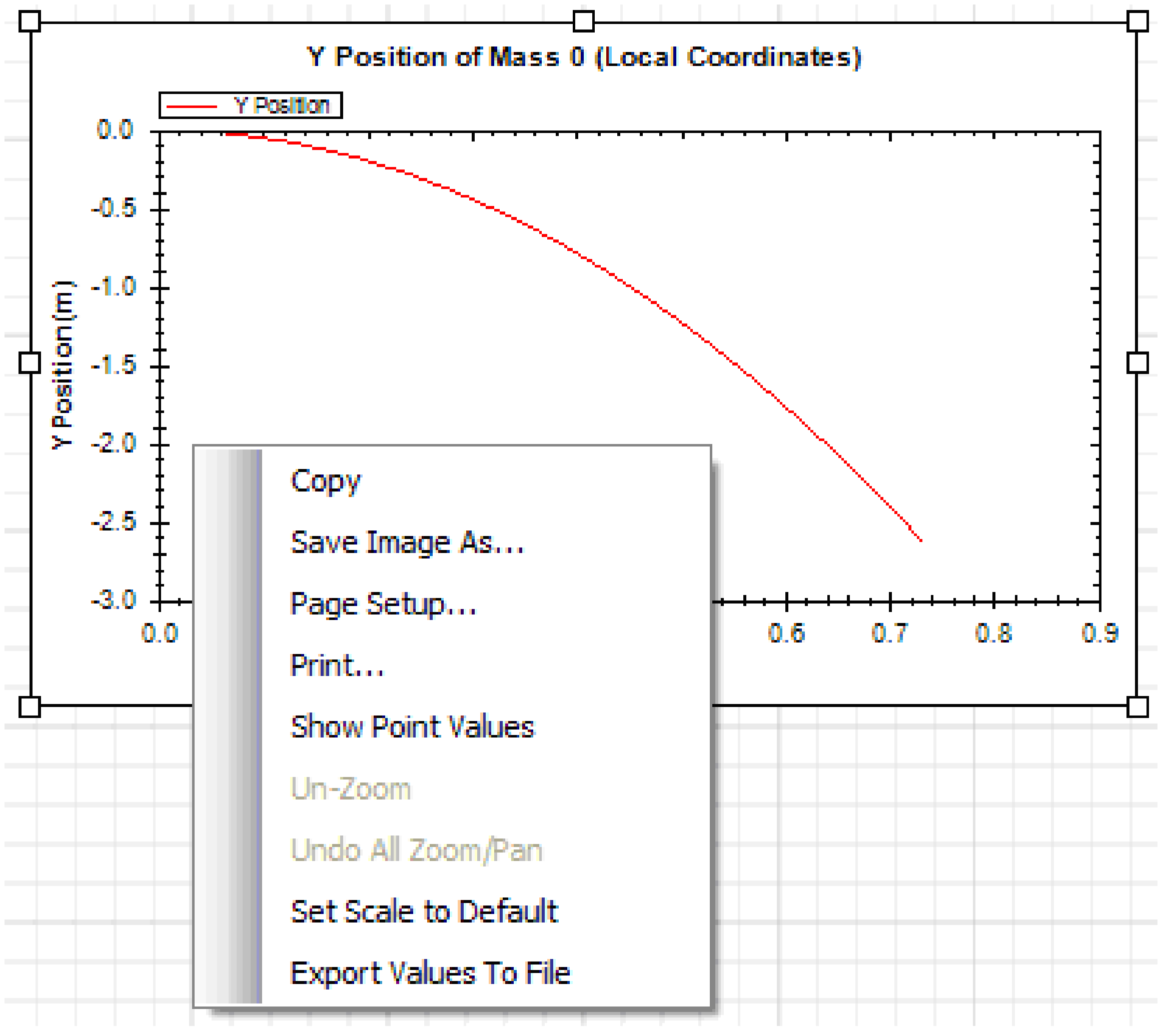


Fig. 11. Additional graph options

C. The Rendering Engine

The rendering engine is in charge of drawing objects on the screen. Because this software is limited to 2D physics, a powerful rendering engine is not needed. The software utilizes GDI+ for its rendering engine, which is the rendering engine used for the Windows XP operating system.

1. IRenderable

Unlike the other engines in our framework, the rendering engine only relies on one interface. The primary interface for displaying an object in the framework is `IRenderable`. If an object implements this interface, then it must define a `Render()` function which can be called to draw the object on the screen. The `IRenderable` interface is extremely simple and is displayed below.

```
public interface IRenderable
{
    void Render(Graphics g, Converter conv);
}
```

How the object is rendered on the screen depends entirely on the object's implementation of the `Render()` function. This makes it easy to render all of the objects in the framework. The program simply loops through the list of objects that implement the `IRenderable` interface and calls each renderable object's `Render()` method.

The Graphics object that is passed to `Render()` is specific to GDI+. If we wanted to switch to a more powerful rendering engine in the future, we would just pass a different drawing context into the `Render()` function and redefine each renderable object's `Render()` function to work with the new rendering engine.

CHAPTER IV

RESULTS

We set out to accomplish a number of goals in this thesis. First, we wanted to design and implement a framework for visual simulation software that can handle most of the systems encountered in an undergraduate dynamics course. The framework should be extensible, making it easy to improve the framework in the future. Second, we wanted to use feedback from user testing to make the software more intuitive and easy-to-use. Finally, we wanted to assess the effectiveness of using the software as a tool for improving students' understanding of dynamics.

A. The Framework

We successfully built a solid object-oriented framework for visual simulation software which is robust enough to use in an undergraduate dynamics course. The framework was implemented using the interfaces and classes described in chapter III. The software contains a set of "building blocks" that the user can utilize to create models of the systems that they encounter in class. This includes masses, supports, springs, dampers, and rods.

1. Framework Flexibility

To be used as an educational tool, the software should be flexible enough to handle most of the systems encountered in an undergraduate dynamics course. With the set of building blocks implemented in the framework, students are able to construct models of the majority of the systems they encounter in class. The software also has additional features that make the simulation more believable.

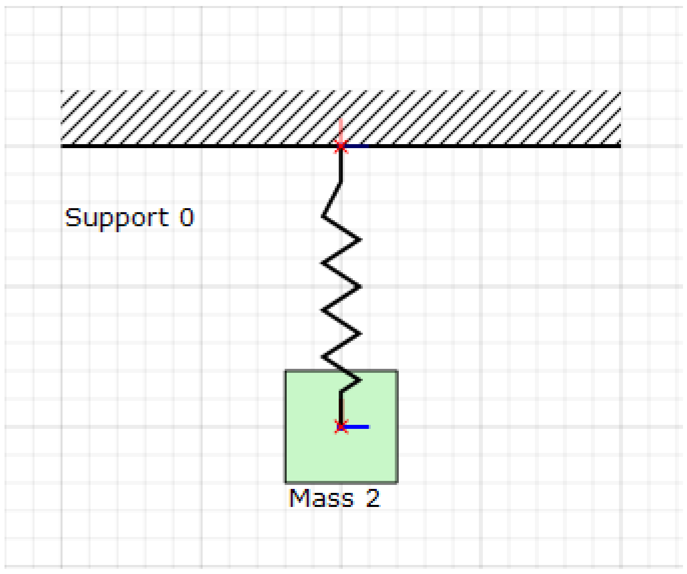


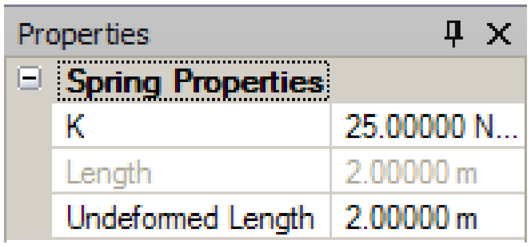
Fig. 12. Spring-mass system

a. Spring-Mass System

A spring-mass system is an example of a system that a student usually encounters in an undergraduate dynamics course. Using the software, a student is able to construct, tweak and analyze a spring-mass system in a matter of minutes. Figure 12 shows a spring-mass system visualized in the software.

This system is built using three building blocks: a support, a mass, and a spring. Using the mouse, the user can easily connect a spring to other objects by clicking and dragging the end points of the spring over a mass or support. When the mouse button is released, the connection is made, and the end point turns into a red “x” to show that it is now connected.

The user can also adjust the properties of objects by clicking on the object. When an object is selected by the user, its properties are displayed in the properties box. The properties box displays the relevant properties of the selected object. Figure 13 shows the properties box when a spring has been selected. The “Length” property is



Properties

Spring Properties	
K	25.00000 N...
Length	2.00000 m
Undefomed Length	2.00000 m

Fig. 13. Spring properties

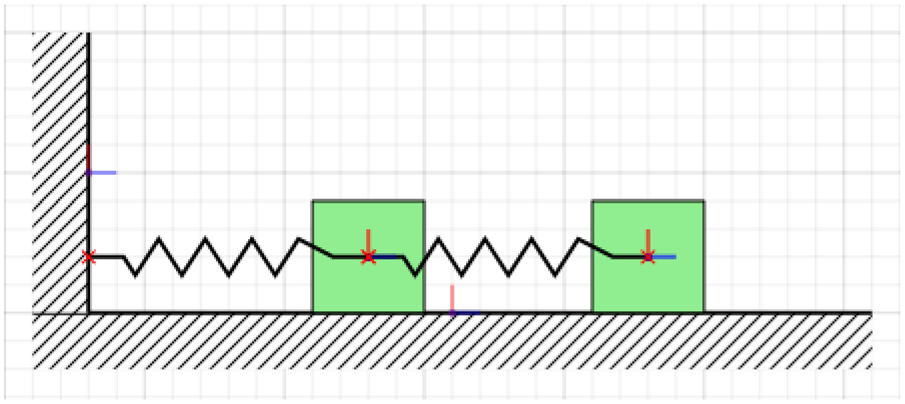


Fig. 14. More complex spring-mass system

grayed out to indicate to the user that it cannot be edited. However, the user can edit the other two properties by entering new values in with the keyboard. Each building block has a unique set of properties that the user can adjust.

A more complex spring-mass system can be created with little additional effort by simply adding and connecting more objects together. Once a user knows how to setup a simple system, they can use that knowledge to build more complicated systems. Figure 14 demonstrates a more complex spring-mass system constructed in the software.

b. Collision Detection and Response

To achieve more realistic simulations, the software includes rudimentary collision detection and response. Because the software only handles point masses at the time, a more robust collision detection and response scheme is not implemented. The software's current collision detection and response has many limitations.

The masses and supports are drawn as boxes and rectangles in the software to emulate the look of the diagrams a student sees in class. However, they are not simulated as rigid bodies. This was a deliberate design choice to accommodate the fact that an undergraduate dynamics course typically starts by using point masses, not rigid bodies.

The discrepancy between what gets drawn and what gets simulated presents an interesting challenge for handling collision detection and response. Based on what is drawn, it may appear as if the mass is rigid body, which has an orientation and the ability to rotate. From the point of the view of physics engine, the mass is simply a point with a position. Similarly, the support is drawn as a rectangle, which could imply that it is a solid static mass. In the simulation system, it is simply seen as a line segment.

Since the physics engine only sees the mass as a point the support as a line segment, the collision detection and response is very limited. A mass will only collide with a support if it hits from the "top" of the support. Figure 15 shows two scenarios of a mass falling and colliding with a support, and it illustrates a limitation of our collision detection and response. Notice how the support on side A is facing away from the mass as it falls. As a result, the mass appears to go through the support when it is simulated. The support on side B is facing opposite to the direction of the mass, so the mass collides with the support as expected.

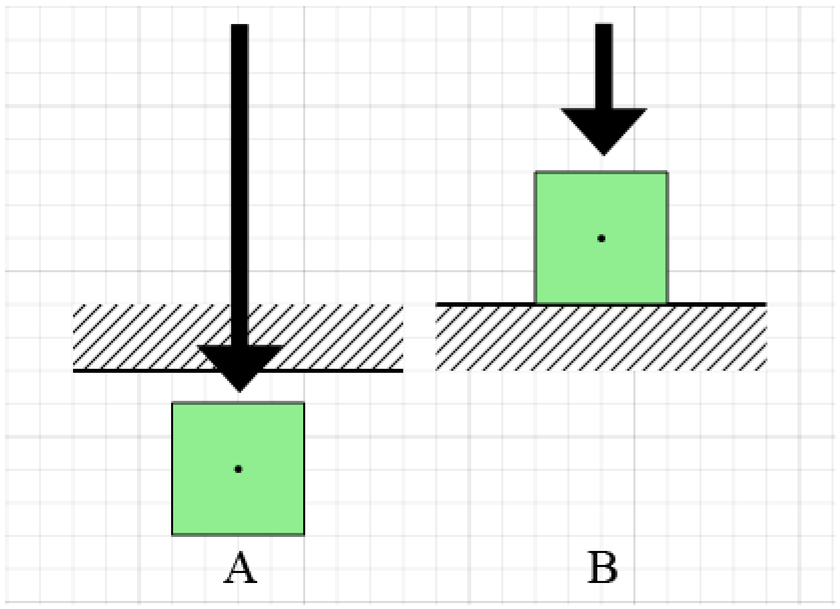


Fig. 15. Collision with supports

Another challenging issue arose when handling collision response. Because the masses are drawn as boxes, one might expect the mass to spin or rotate after colliding with a support. Since the masses are modeled as point masses in the simulation system, this is not possible. The software does a couple of things to get around this issue. First, the collisions are non-elastic, meaning a mass will not bounce when it collides with a support. Second, if a mass collides with a rotated support, the box is oriented to align with the support.

Figure 16 shows a mass before and after it collides with a support that has been rotated. Notice how the mass sticks to the support after it has collided. While this may not be a physically correct solution, it solves the issue of collision response from a visual perspective.

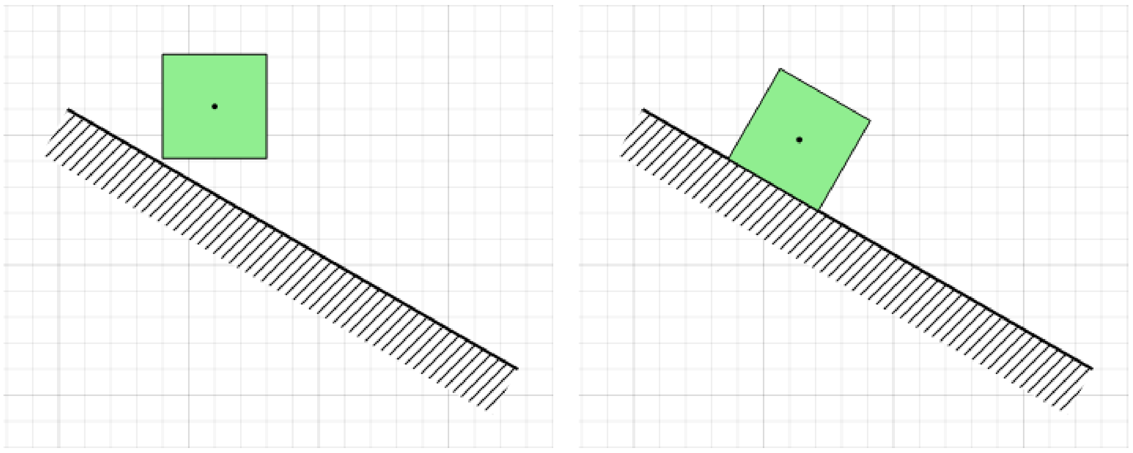
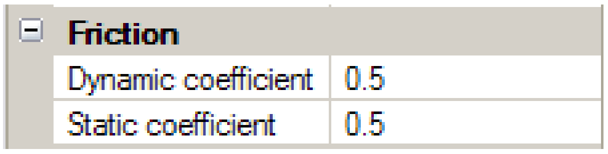


Fig. 16. Collision with a rotated support



Friction	
Dynamic coefficient	0.5
Static coefficient	0.5

Fig. 17. Friction properties for a support

c. Friction

Friction is also implemented in the software to simulate realistic behavior when a mass comes into contact with a support. After the mass collides with the support in Figure 16, one would expect it to slide down the support and stop or fall off, depending on the amount of friction between the objects. The user is able to control the amount of friction by selecting a support and setting the dynamic coefficient of friction and the static coefficient of friction as shown in Figure 17.

2. Framework Extensibility

Another major goal of this thesis was to create a strong object-oriented framework to make it easily extensible. Using the classes and interfaces described in chapter III, a modular and extensible framework was implemented. During the development of the software, additional building blocks and features were added to the framework, validating the adaptability of the framework.

a. Motors

Motors were added to the software after the initial set of building blocks were already functioning. A motor is an object that can be attached to a mass. When a motor is attached to a mass, it applies a user-defined, time-varying force to the mass during the simulation. A motor represented by an orange arrow as shown in Figure 18. The arrow points in the direction of the applied force.

To add the `Motor` object into the framework, we created two new classes. The first class, `LocalForce`, is an abstract class that can be used for any object that applies a force and can be attached to a single object. Next, we added the `Motor` class which inherits from `LocalForce`. The `Motor` class implements the `IForceable` interface, so it must define an `ApplyForce()` function. For a motor, this function uses the current time to calculate the force it is generating at that time and applies this force to the force accumulator of the object it is connected to.

Figure 19 shows the properties that the user can change when a motor is selected. The "Type" property allows the user to choose between a sine and cosine function. The "Angle" property enables the user to specify the direction of the force. The remaining properties can be used to adjust the amplitude and frequency of the time-varying force function.

b. Rods

Rods were another building block added to the framework after the initial building blocks were implemented. A rod maintains a fixed distance between two objects.

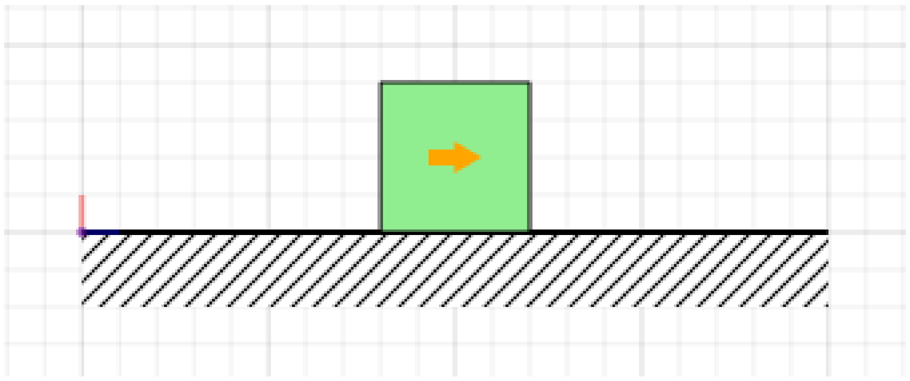
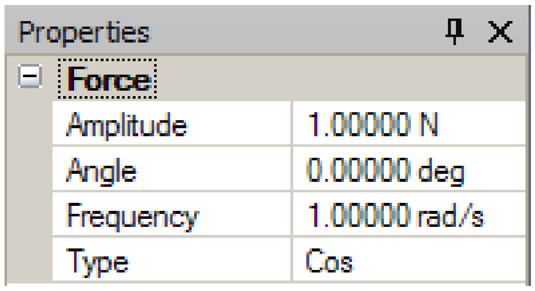


Fig. 18. Motor attached to a mass



Properties

Force

Amplitude	1.00000 N
Angle	0.00000 deg
Frequency	1.00000 rad/s
Type	Cos

Fig. 19. Motor properties

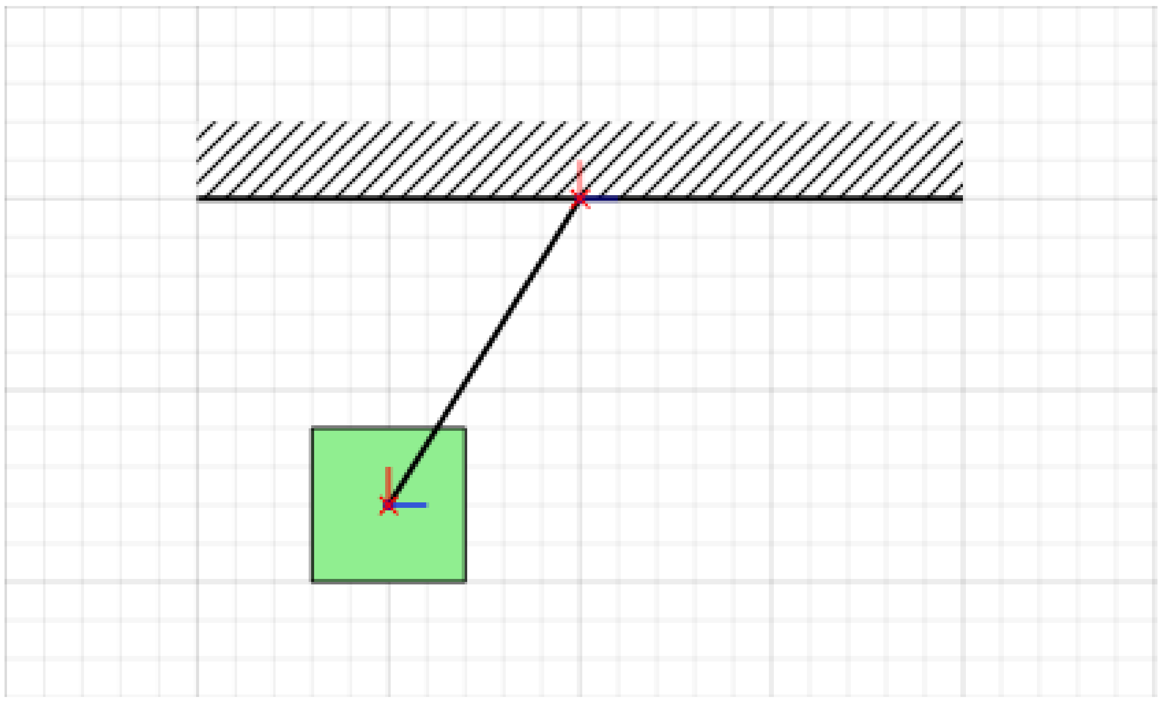


Fig. 20. Rod connecting a mass and support

Similar to springs and dampers, a rod can be attached to masses and supports. Figure 20 demonstrates an example of how a rod can be used in a system to model a pendulum.

To add rods into the framework, we implemented the constraint force method described in [8] and [9]. Using this method, we define constraints and then compute constraint forces which, added to the regular applied forces, cause the system to accurately satisfy the defined constraints. In the context of our framework, a distance constraint is defined for each rod in the system. During each step of the simulation, constraint forces are computed for each rod and these forces are applied to the attached objects. As a result, objects connected by a rod will remain a fixed distance from each other throughout the simulation.

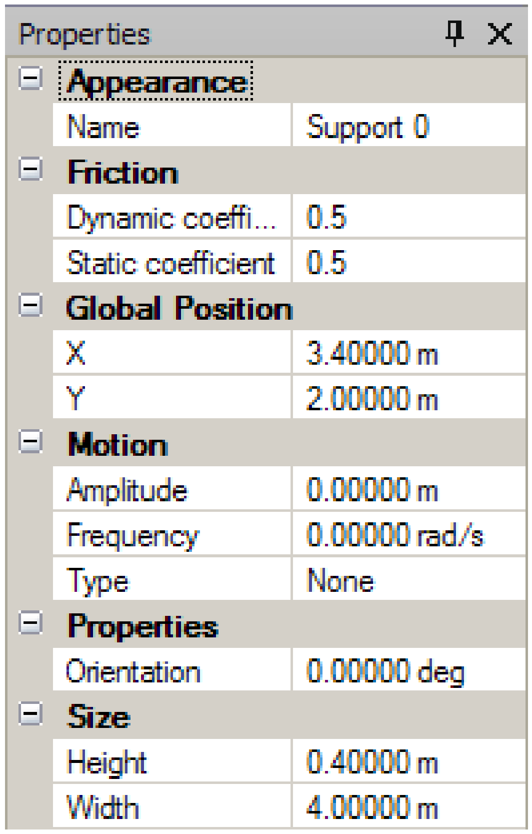
Because a rod can be connected to two objects, the `Rod` object derives from the `Connector` class described in chapter III. However, since constraint forces not only depend on the current state but also other applied forces, a new function called `ComputeConstraintForces()` was created. This function is called in the `SystemDynamics()` function after the regular applied forces have been computed. It computes constraint forces for each rod in the system and adds them to the force accumulators of the appropriate objects.

c. Support Motion

Ground motion is an important concept that students typically encounter in an undergraduate dynamics course. While a support could be used to model the ground in the initial version of the software, there was no way to make the support move to simulate ground motion. To fix this issue, we extended the `Support` object by adding the support motion feature. This feature gives the user the ability to define a function that moves the support over the time.

To implement this feature, an `ApplyMotion()` function was added to the `Support` object. This function takes the current simulation time and moves the support to the correct position based on the user-defined motion function. `ApplyMotion()` is called at the beginning of the `SystemDynamics()` function to ensure that the supports are in the correct position before the forces in the system are computed.

Figure 21 shows the controls that the user has over the support motion function. The "Type" property allows the user to choose between a sine and cosine function or turn the motion off. The "Amplitude" and "Frequency" properties enable the user to adjust the amplitude and frequency of the sine or cosine function.



Properties

Appearance	
Name	Support 0
Friction	
Dynamic coeffi...	0.5
Static coefficient	0.5
Global Position	
X	3.40000 m
Y	2.00000 m
Motion	
Amplitude	0.00000 m
Frequency	0.00000 rad/s
Type	None
Properties	
Orientation	0.00000 deg
Size	
Height	0.40000 m
Width	4.00000 m

Fig. 21. Support properties

B. Usability

Another major goal of this thesis was to create software that is intuitive and easy-to-use. User testing is essential in the process of creating a usable product. The software developed in this thesis was tested by actual students, providing very helpful feedback on the usability of the software. Based on this feedback, a variety of improvements were made to the software. These ranged from functional improvements to visual enhancements that helped improve the overall usability of the software.

1. User Testing

Two rounds of user testing were conducted to get feedback on the software. The purpose of this feedback was to identify aspects of the software that needed to be improved to make it easier for the students to use it as a learning tool.

For both rounds of testing, the users consisted of students currently enrolled in an undergraduate dynamics course in the civil engineering department. These user testing sessions were held in a computer lab with fifteen computers. Each round of testing included thirty students split into two sessions, because of the constraints of the computer lab. During the first half of each session, students were allowed to experiment with the software to get familiar with the interface. For the latter half of the session, the students were given problems that they had encountered in class and were asked to set these problems up using the software. Throughout the session, we were able to walk around the computer lab and observe how the students were using the software. Students were also able to ask us for help when they ran into issues with the software. At the end of each session, the students filled out an anonymous questionnaire where they were asked to leave their feedback on the software and suggest improvements. A short open discussion was also held at the end of each session where students could verbally express their opinions and suggestions for the software.

The observations and feedback from the user testing sessions were used to help identify aspects of the software that needed improvement. The testing sessions exposed the need for certain features such as a snappable grid to enable the user to precisely layout systems in the software. The testing sessions also aided in prioritizing enhancements to the software that would make it more effective and easier to use. In the following sections, there are more detailed explanations of how the feedback from user testing was used to make improvements to the software.

2. Visual Hints

To make the visual simulation software more intuitive, visual hints were employed so that the user would know about the current state of the objects by simply looking at the screen. Through the use of visual properties like color and line style, the software strives to keep the user informed of the current state of the objects at all times. Dialog boxes are used to notify the user when an action must be performed before they can proceed. The following subsections will detail some of the visual hints employed in the software.

a. Detached Connectors

As previously described in chapter III, a connector object must be connected to two connectable objects to properly function. However, the user may not be aware of this. If the connector appears exactly the same whether or not it has been connected, then the user may not realize that it is not connected. To avoid this problem, the software differentiates a detached connector from an attached connector by altering the color and line style.

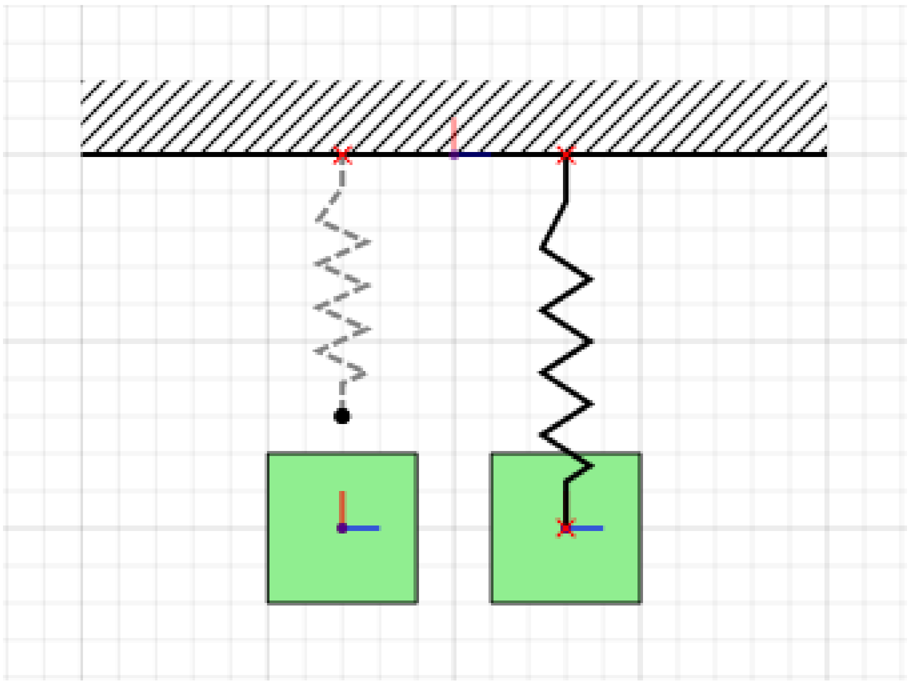


Fig. 22. Detached spring and connected spring

Figure 22 illustrates the visual difference between a detached spring and a connected spring. The detached spring is drawn with a dashed line in a gray color to indicate that it is not properly connected. This immediately informs the user that the spring is in a detached state and clearly distinguishes between the two possible states for a spring.

For consistency, all connector objects follow the same convention. Figure 23 illustrates this by showing a detached damper and a connected damper. Once again, it is easy for the user to determine the state of the damper by simply looking at how it is drawn on the screen. This is extremely useful for the user, because the simulation cannot be run unless all of the connectors have been properly connected.

If the user tries to run the simulation when a connector is not attached properly, a warning message will pop up as shown in Figure 24. This message tells the user that one or more objects need to be connected before the simulation can run. Because the

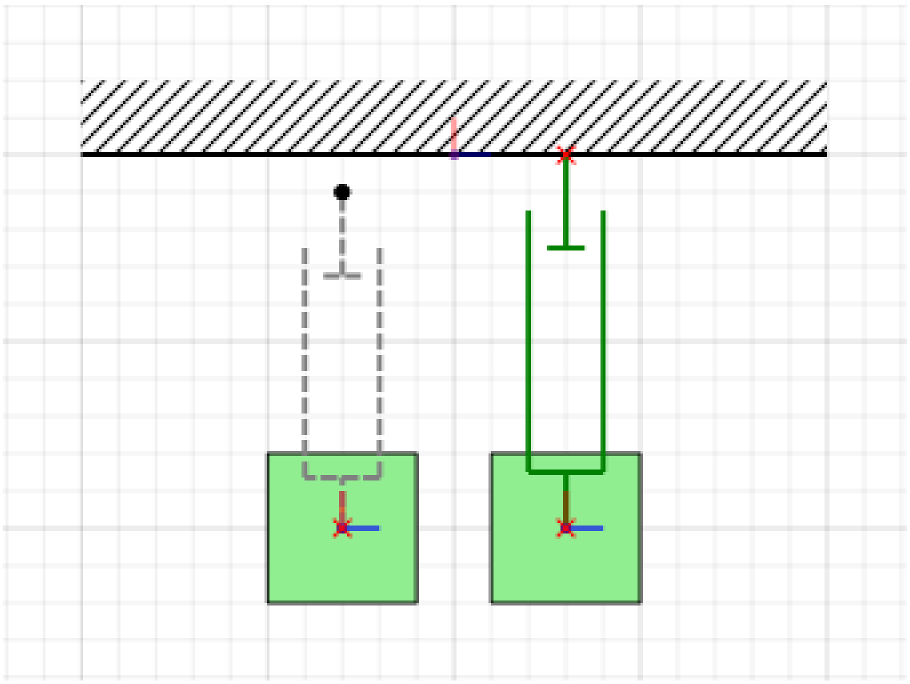
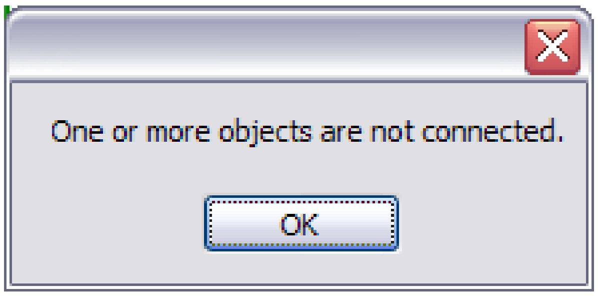


Fig. 23. Detached damper and connected damper

detached connectors are distinctly rendered, the user can easily identify what needs to be connected.

b. Color-coded Springs

To keep the user up to date on the state of a spring, the springs are color-coded. A spring is rendered in different colors, depending on the current length of the spring.



One or more objects are not connected.

Fig. 24. Warning message

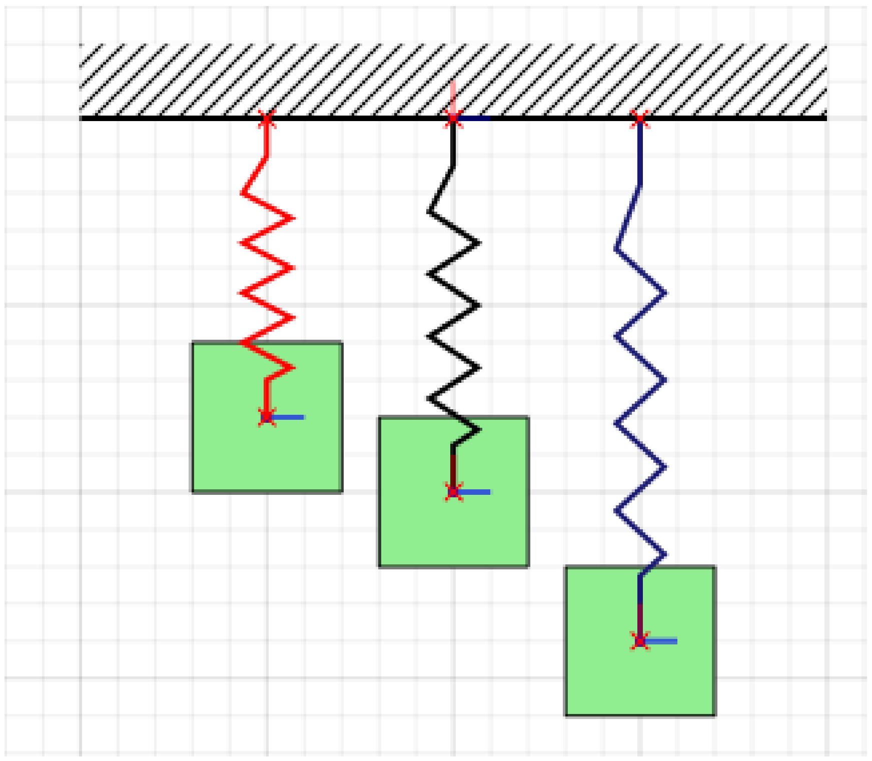


Fig. 25. Color-coded springs

When a spring is compressed, it appears red. A spring at its rest length appears black. Lastly, a spring that is stretched appears blue.

Figure 25 demonstrates how three springs with the same rest length are rendered when they are at different lengths. The spring on the left has been compressed, so it is red. The center spring has not been moved, so it is black. The spring on the right has been stretched down, so it is blue.

This simple visual hint makes it possible for the user to know what state a spring is in at all times. This is useful for setting up a system or analyzing the system when the simulation is paused. It also removes the need to dig into the spring's properties to determine whether the spring is stretched or compressed.

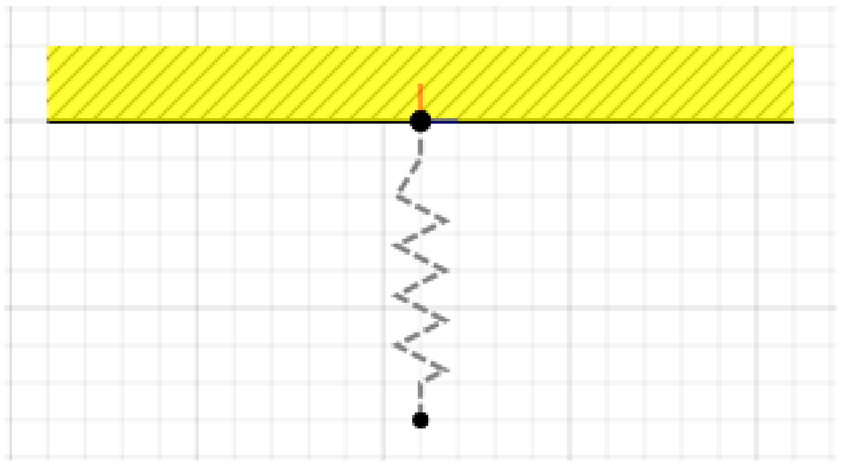


Fig. 26. Object highlighting

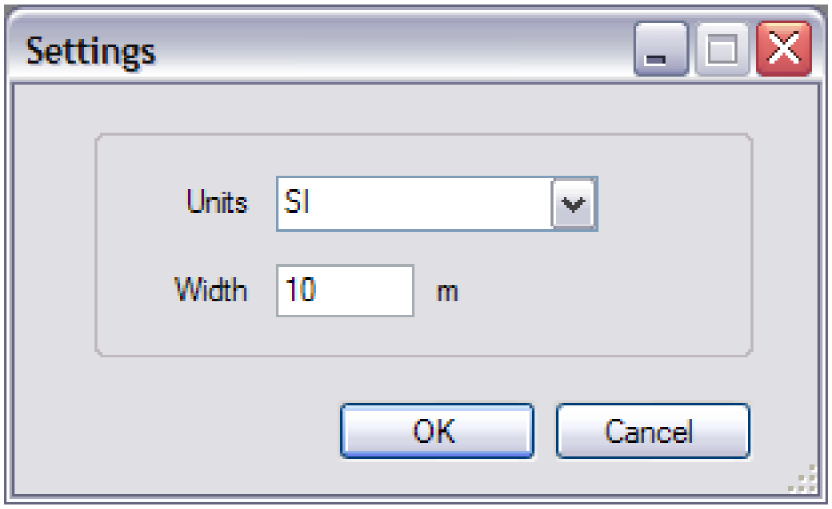
c. Highlighting

Figure 26 demonstrates a visual hint that lets the user know when an object is connectable. When the user drags a connection point from the end of a spring over a mass or support, the mass or support gets highlighted. This informs the user that a connection can be made if they drop the point there. The object will only remain highlighted if the connection point is hovering over it. As soon as the connection point is moved away, the highlighted object is drawn normally again.

This visual hint makes it easy for users to identify which objects they can attach a connector to. It also serves as a notice to the user that they are about to make a connection between two objects. This prevents the user from accidentally attaching two objects without realizing it.

3. Initial Conditions

The first step to solving any dynamics problem is specifying the initial conditions. In the context of an undergraduate dynamics course, students are often given the initial conditions in the problem statement. This can include specifications such as the unit



Units SI

Width 10 m

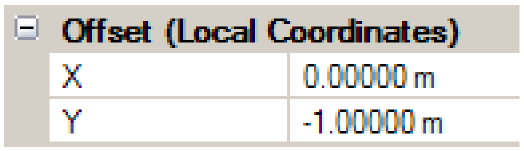
Fig. 27. Initial settings

system, the initial position of an object, and the initial velocity of an object. During the user testing sessions, students had difficulty specifying the initial conditions for the systems they were trying to build. To address this issue, the software attempts to simplify the input of the initial conditions by prompting to the user to enter initial settings and giving the user the ability to modify local coordinate systems for each object.

a. Initial Settings

Regardless of the problem at hand, there are certain settings that the user has to define for every system that they build. Two settings that are essential for any system are the unit system and the size of the canvas. To ensure that the user sets these properties, a dialog box, as shown in Figure 27, pops up every time the user creates a new file in the program. The user can choose the unit system and define the maximum width needed for the canvas.

After the user clicks "OK", the file is created with the specified settings. This



Offset (Local Coordinates)	
X	0.00000 m
Y	-1.00000 m

Fig. 28. Offset from local coordinate system

dialog box encourages the user to think about these initial conditions before they even start constructing the system in the program. In case the user happens to make a mistake, the user can switch between unit systems at a later time.

b. Local Coordinate Systems

Each connectable object has its own local coordinate system in the software. The user can select it and move it around the canvas like any other object. While the local coordinate system does not affect the behavior of the simulation, it is a crucial tool for helping the user set the initial conditions of a system.

Local coordinate systems are valuable to a user when they are trying to setup a problem where the initial conditions are given as an offset from a certain position. For example, a problem might state that a spring is initially stretched 1 meter from its undeformed length. Assuming that the spring is at its rest position, the user can select the mass to which it is attached to access the mass' properties. The user can then enter an offset from the local coordinate system for the mass as shown in Figure 28. The visual result of entering the offset from the local coordinate system is shown in Figure 29.

4. Exporting Data

While the software allows the user to graph and view values associated with objects, the program is not very flexible when it comes to data analysis. Since there are already

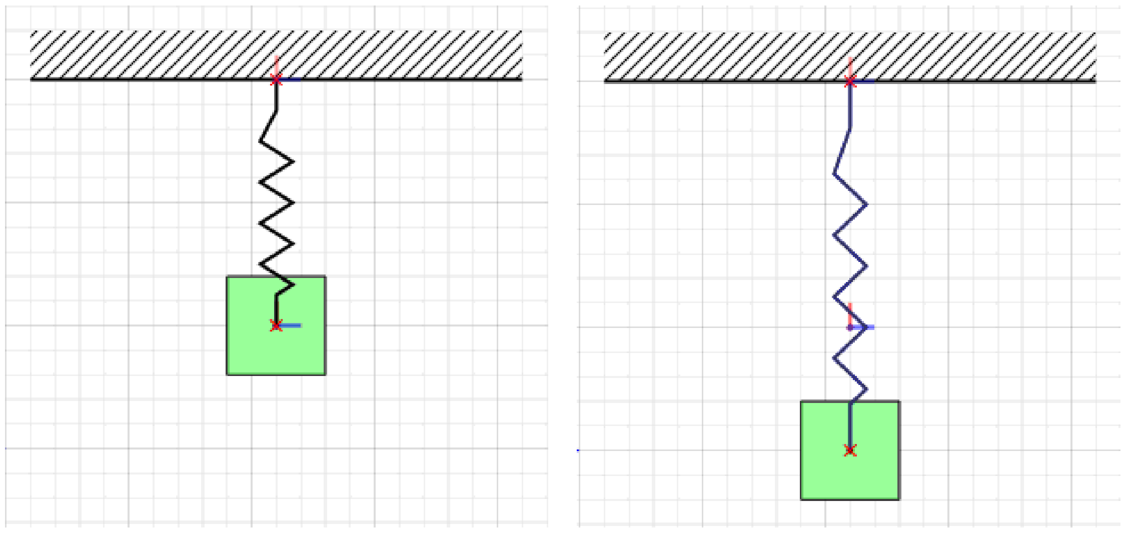
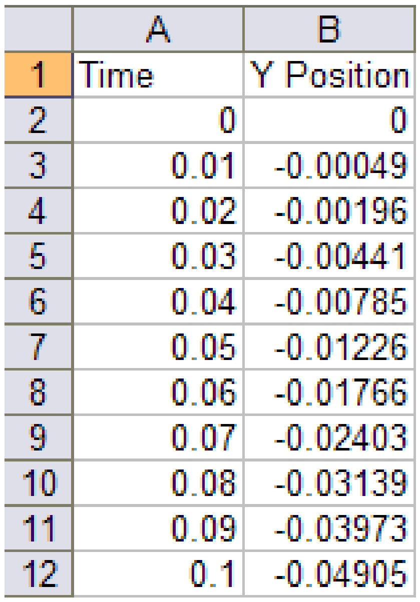


Fig. 29. Visual display of offset from local coordinate system

programs, like Microsoft Excel, that are well suited for data analysis, we added a feature that allows the user to export the simulation data to an external program rather than implementing data analysis tools within the software. In addition to reducing development time, relying on a program like Excel for data analysis makes it easier for the user, because most undergraduate students are already familiar with using a program like Excel.

As previously shown in Figure 11, the user has the option to “Export Values To File” when a graph is right-clicked. When this option is selected, the user chooses a name and location for the file to be saved. The data is then exported as a comma-separated values (CSV) file, which can be opened in Excel for more detailed analysis. Figure 30 shows an example of how the exported values from the program appear when the file is opened in Microsoft Excel.



	A	B
1	Time	Y Position
2	0	0
3	0.01	-0.00049
4	0.02	-0.00196
5	0.03	-0.00441
6	0.04	-0.00785
7	0.05	-0.01226
8	0.06	-0.01766
9	0.07	-0.02403
10	0.08	-0.03139
11	0.09	-0.03973
12	0.1	-0.04905

Fig. 30. Comma-separated values in Excel

5. User Documentation

With any software, there is expected to be a learning curve involved as the user gets familiar with the program. During the user testing sessions, the students often asked similar questions about the functionality of the software and how to use it. Based on this feedback, we decided that the best solution was to consolidate the answers to these common questions on a website. In addition to answering frequently asked questions, the website also includes user documentation and tutorials, providing users with a resource that is always available when they run into a roadblock while using the software.

Figure 31 displays a screenshot of a step-by-step tutorial on how to create a spring-mass system in the software. This basic tutorial walks users through the basics of constructing a simple system in the software. After completing this tutorial, the user will have the basic skills required to construct more complicated systems. Additional tutorials teach the users how to use more advanced features like support motion.

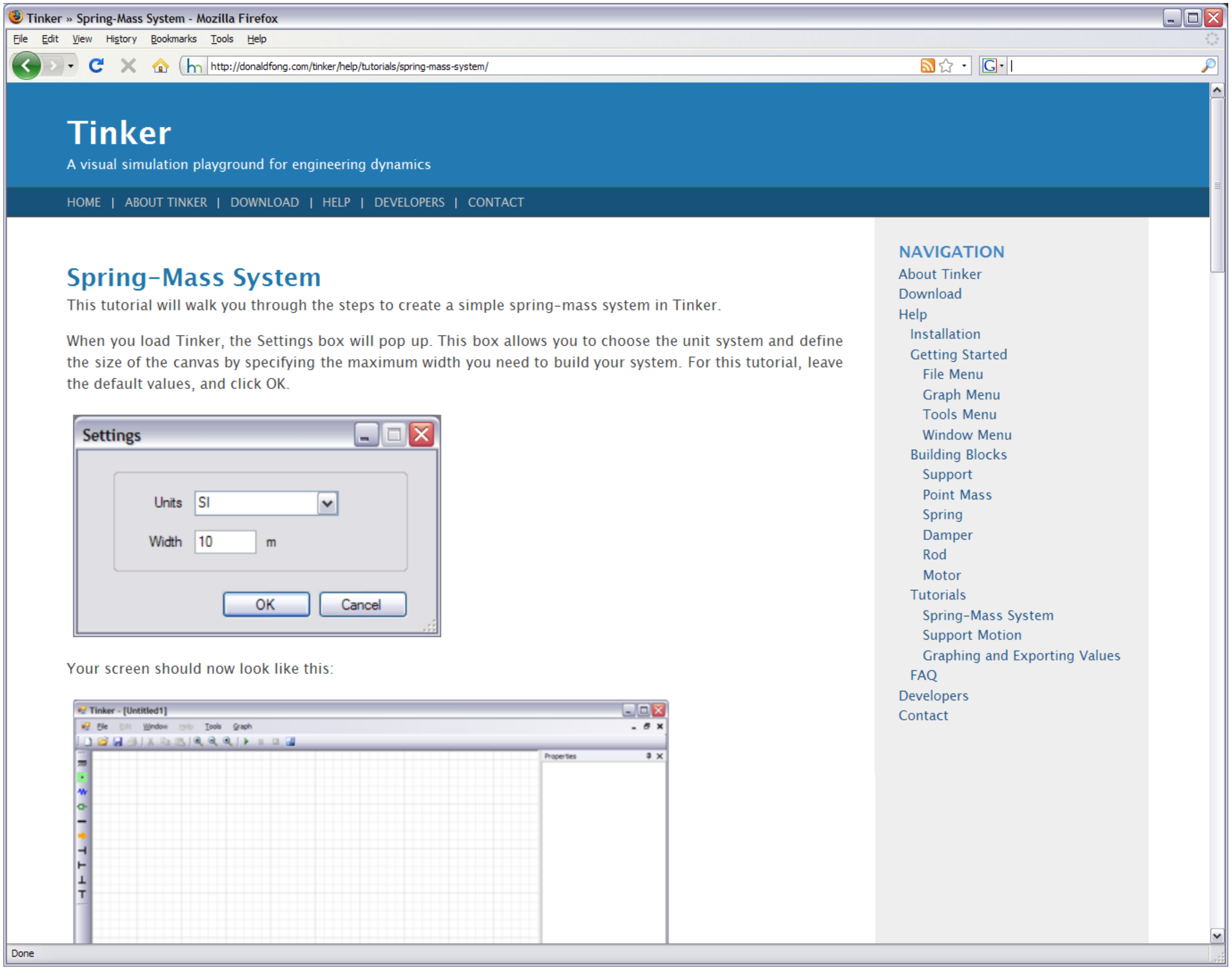
C. Impact on Student Learning

One of the goals of this thesis was to evaluate the effectiveness of using the software to help improve student learning of dynamics concepts. To measure the impact of the software on student learning, we collected feedback from the students at the end of the user testing sessions. We also analyze results from the Student Assessment of Learning Gains (SALG) instrument for the spring 2007 semester of an undergraduate dynamics course. SALG is a web-based instrument consisting of statements about the degree of "gain" (on a five-point scale) which students perceive they have made in specific aspects of the class.

At the end of each user testing session, we held an open discussion where the students could discuss issues they ran into while using the software and suggest improvements. Most of the issues that students reported while using the software relate to the user interface. While students quickly learned how to connect objects, the process of setting up an entire system proved to be very challenging. Students struggled to set the initial conditions of the systems they were trying to build, because they were unsure of how to change object properties and unaware of some features that were available to them.

In the SALG survey, students were asked a series of questions about the software, referred to as Tinker (Table 1). The first few questions asked the students to rate how much the software helped them in the context of the course on a five-point scale. The last question was open ended to allow students to comment on what they did and did not like about the software.

The SALG results from the spring 2007 semester indicate that students did not



Tinker » Spring-Mass System - Mozilla Firefox

http://donaldfong.com/tinker/help/tutorials/spring-mass-system/

Tinker

A visual simulation playground for engineering dynamics

HOME | ABOUT TINKER | DOWNLOAD | HELP | DEVELOPERS | CONTACT

Spring–Mass System

This tutorial will walk you through the steps to create a simple spring–mass system in Tinker.

When you load Tinker, the Settings box will pop up. This box allows you to choose the unit system and define the size of the canvas by specifying the maximum width you need to build your system. For this tutorial, leave the default values, and click OK.

Settings

Units: SI

Width: 10 m

Your screen should now look like this:

NAVIGATION
- About Tinker
- Download
- Help
  - Installation
  - Getting Started
    - File Menu
    - Graph Menu
    - Tools Menu
    - Window Menu
  - Building Blocks
    - Support
    - Point Mass
    - Spring
    - Damper
    - Rod
    - Motor
  - Tutorials
    - Spring–Mass System
    - Support Motion
    - Graphing and Exporting Values
  - FAQ
- Developers
- Contact

Fig. 31. Tutorial for spring-mass system

Table I. SALG Questions

How much do you agree that the use of Tinker helped tie course concepts together?
How much do you agree that the use of Tinker helped clarify the mathematical model?
How much do you agree that Tinker helped you visualize the response?
Any specific comments on Tinker?

feel the software was effective at improving their understanding of dynamics. On average, students did not feel that the software helped to tie course concept together, clarify the mathematical model, or visualize the response of systems. A detailed summary of the statistics from the SALG results is presented in Appendix A.

Based on the comments from the students, many expressed frustration with the usability of the software in its current state. Students felt that the user interface did not facilitate the process of modeling a system for their homework problems. They also expressed difficulty in figuring out how to set the initial conditions of a system. Improvements to the user interface are the key to making the software a more effective learning tool.

CHAPTER V

CONCLUSION AND FUTURE WORK

This thesis details and describes the design of an extensible framework for software that allows students to visually construct, simulate, and analyze systems typically encountered in an undergraduate dynamics course. The framework supports point masses and supports with collision detection, resting, and friction. Springs, dampers, and rods can be used to connect masses and supports to each other. Users can graph values, save files, and export data to use in other software for more advanced analysis.

During the development of the framework, new building blocks were easily added to the framework. The extensibility of the framework was verified by the addition of the Motor and Rod objects into the existing set of building blocks. The new building blocks were integrated into the framework's object hierarchy without difficulty. The object-oriented design of the framework enables additional building blocks, such as pulleys, to be integrated with ease.

Two rounds of user testing were conducted in which students currently enrolled in an undergraduate dynamics course tested the software. These testing sessions were used to identify ways to enhance the functionality of the software. Based on the feedback from these sessions, features such as visual hints, tools to set initial conditions, and tutorials were implemented to improve the usability and ease of use of the software.

Student feedback and SALG results were analyzed to assess the effectiveness of using the software as a tool to improve student understanding of dynamics concepts. The analysis showed that the software in its current state is not as effective as we would have liked when it comes to improving student learning. However, the framework provides a solid foundation that can be built upon to create a more effective learning tool because of its modularity and extensibility.[1]

Compared to existing 2D simulation software, the physics engine in our software is rudimentary. The software does not have support for rigid bodies, joints, torsional springs, or pulleys. Adding more advanced dynamics to the framework will help to make the software a more useful learning tool, because students will be able to model more systems. Since the framework is modular, an existing 2D physics engine could be integrated into the framework to save development time. Improvements are also needed in the collision detection and response of objects. This will result in more realistic behavior when more complex systems are simulated.

The user interface can be improved to increase the usability of the software. Although the point and click interface of a mouse is ubiquitous, it may not be the best method of user input for this software. An alternative method of input, such as the sketch-based interface developed in [10], could make the software more intuitive and natural for students to use.

[1]A newer version of the software has already been developed, and preliminary results are very promising.

REFERENCES

[1] D. L. Evans, G. L. Gray, S. Krause, J. Martin, C. Midkiff, et al. “Progress on Concept Inventory Assessment Tools,” in *Proceedings of 33rd Frontiers in Education Conference*, Boulder, CO, Nov. 5-8, 2003, pp. T4G-1–T4G-8.

[2] L.C. McDermott. “Improving Student Learning in Science,” *LTSN (Learning and Teaching Support Network) Physical Sciences News*, vol. 4, no. 2, 2003, pp. 6-10.

[3] L. R. Barroso, J. Morgan, and N. Simpson, “Active Demonstrations for Enhancing Learning,” in *Proceedings of 37th ASEE/IEEE Frontiers in Education Conference*, Milwaukee, WI, Oct. 10-13, 2007, pp. S2A-1–S2A-5.

[4] K. Gramoll. “Using ‘Working Model’ to Introduce Design into a Freshman Engineering Course,” in *Proceedings of 1994 ASEE Conference*, Edmonton, Alberta, Canada, June 26-29, 1994, pp. 1628-1633.

[5] R. E. Flori, M. A. Koen, and D.B. Oglesby. “Basic Engineering Software for Teaching (“BEST”) Dynamics,” *Journal of Engineering Educations*, January 1996, vol. 85, no. 1, pp. 61-67.

[6] T. Naps, S. Cooper, B. Koldehofe, C. Leska, G. Roessling, et al. “Evaluating the Educational Impact of Visualization,” in *Proceedings of ITiCSE-WGR ’03: Working Group Reports from ITiCSE on Innovation and Technology in Computer Science Education*, Thessaloniki, Greece, 2003, pp. 124-136.

[7] D. Sandin, P. Trunfio., L. Yaeger, P. Hickman, and R.S. Wolff. “Visualization technologies as a tool from science education,” in *Proceedings of SIGGRAPH ’90*, Dallas, TX, 1990, pp. 1301-1316.

[8] D. Baraff, A. Witkin, and M. Kass. “Physically Based Modeling,” *SIGGRAPH ’99 Course Notes*, 1999, pp. B1-B8, F1-F12.

[9] A. Witkin, M. Gleicher, and W. Welch. “Interactive Dynamics,” *Computer Graphics*, vol. 24, no. 2, 1990, pp. 11-21.

[10] C. Alvarado. “A Natural Sketching Environment: Bringing the Computer into Early Stages of Mechanical Design,” M.S. Thesis, MIT, Cambridge, MA. 2000.

APPENDIX A

SALG RESULTS

How much do you agree that the use of Tinker helped to tie course concepts together?

1 Strongly agree	0% (0)
2 Agree	12% (5)
3 Neutral	19% (8)
4 Disagree	33% (14)
5 Strongly disagree	37% (16)
Average = 3.95, S.D. = 1.01, N = 43	

1 Strongly agree	2% (1)
2 Agree	15% (8)
3 Neutral	19% (10)
4 Disagree	37% (19)
5 Strongly disagree	27% (14)
Average = 3.71, S.D. = 1.08, N = 52	

How much do you agree that the use of Tinker helped to clarify the mathematical model?

1 Strongly agree	0% (0)
2 Agree	14% (6)
3 Neutral	21% (9)
4 Disagree	30% (13)
5 Strongly disagree	35% (15)
Average = 3.86, S.D. = 1.05, N = 43	

1 Strongly agree	2% (1)
2 Agree	15% (8)
3 Neutral	19% (10)
4 Disagree	37% (19)
5 Strongly disagree	27% (14)
Average = 3.71, S.D. = 1.08, N = 52	

How much do you agree that Tinker helped you visualize the response?

1 Strongly agree	5% (2)
2 Agree	26% (11)
3 Neutral	24% (10)
4 Disagree	17% (7)
5 Strongly disagree	29% (12)
Average = 3.38, S.D. = 1.27, N = 42	

1 Strongly agree	6% (3)
2 Agree	21% (11)
3 Neutral	27% (14)
4 Disagree	27% (14)
5 Strongly disagree	19% (10)
Average = 3.33, S.D. = 1.17, N = 52	

VITA

Name:

Donald Brian Fong

Address:

Department of Architecture

Langford C418

Texas A&M University

3137 TAMU

College Station, TX 77840-3137

Email Address:

dfong@viz.tamu.edu

Education:

B.S., Computer Science, Northwestern University, 2004

M.S., Visualization Sciences, Texas A&M University, 2008