Hello, M.U.P.P.E.T.S.: Using a 3D Collaborative Virtual Environment to Motivate Fundamental Object-Oriented Learning

Christopher Egert
Interactive Media Group
Information Technology Department
Rochester Institute of Technology
102 Lomb Memorial Drive
Rochester, NY 14623
+1 585 475 4873
cae@it.rit.edu

Kevin Bierre
Enterprise Computing Group
Information Technology Department
Rochester Institute of Technology
102 Lomb Memorial Drive
Rochester, NY 14623
+1 585 475 5358
kjb@it.rit.edu

Andrew Phelps
Interactive Media Group
Information Technology Department
Rochester Institute of Technology
102 Lomb Memorial Drive
Rochester, NY 14623
+1 585 475 6758
amp@it.rit.edu

Phil Ventura
Department of Natural Sciences, Math & Computer Science
St. Thomas University
16401 NW 37th Avenue
Miami Gardens, FL 33054
+1 305 628 6538
pventura@stu.edu

Abstract

With the advent of the objects-first approach for introductory programming, instructors are challenged to think differently regarding the projects and exercises they create for their classrooms. The objects-first approach reduces the emphasis on syntax and encourages the student to focus upon the proper construction and use of classes. This change in emphasis means that students must understand the relationships between classes within a code solution and how such relationships affect the overall design of a system. Unfortunately, such critical thinking exercises can prove challenging to the introductory student, especially if presented in an abstract manner. In this paper, the authors examine how fundamental principles such as inheritance, composition, and association can be conveyed to introductory programming students within a collaborative virtual environment. The examples chosen follow established guidelines for objects-first examples while leveraging features of an engaging, three-dimensional interactive environment.

Categories and Subject Descriptors K.3.2 [**Computers and Education**]: Computer and Information Science Education

General Terms Experimentation, Human Factors, Languages, Theory.

Keywords Programming Education, Virtual Worlds, Graphics.

1. Introduction

In recent years, Sun Microsystem's Java programming language [10] has become the *lingua franca* within freshman programming courses [5]. The selection of Java has also prompted instructors to search for better techniques to promote object-oriented pedagogy within the classroom [1-3,8,11]. The momentum generated by object-oriented approaches has encouraged the ACM/IEEE Task Force for Computing Curriculum 2001 in Computer Science [1] to recommend an object-oriented introductory course sequence as a viable alternative to imperative approaches. The object-oriented sequence places emphasis upon concepts such as objects and inheritance over traditional control structures such as conditionals and loops. Some instructors embrace an even stronger definition of object-oriented pedagogy, which they refer to as "objects-first" [2,3,8]. The objects-first philosophy stresses exposure to concepts such as classes versus instances, methods versus properties, and class relationships such as composition, association, and inheritance within the first few weeks of the introductory course [2,3,11].

At present, there are some reports that objects-first approaches are working within the classroom [11]. However, these same practitioners recognize that there is more to an objects-first approach than just the introduction of abstract concepts such as classes and class relationships. In order to ground abstract concepts, concrete examples must be provided that draw upon the background and experiences of the student. In addition, examples must align with the student perception of computing, which has been influenced by video games and other multimedia, interactive experiences [7]. This paper examines the use of a collaborative virtual reality environment to motivate the exploration of early objects-first concepts such as differentiation between classes and objects, as well as critical concepts such as inheritance, composition, and association.

2. The Graphical Design-Centric Objects-First Approach

The graphical, design-centric, objects-first approach discussed within this paper was developed by one of the authors along with others at University at Buffalo, SUNY [2,3,11], and was inspired by the work started at Brown University [6]. The approach meets the criteria set forth by CC2001 for objects-first curricula and embraces the concept that object-oriented programming and design should be thematic in an object-oriented course. While CC2001's definition requires objects and inheritance to be covered prior to control structures, the graphical, design-centric objects-first course goes further. As polymorphism plays as central role in object-oriented programming, it is likewise covered prior to selection and iteration. In addition, design patterns are introduced just-in-time for solving design problems encountered.

To foster both a vocabulary for design as well as an appreciation for the specification of a design independent of the programming

OOPSLA'06 October 22–26, 2006, Portland, Oregon, USA.
ACM 1-59593-491-X/06/0010.

language used, a simplified version of UML class diagrams is introduced.

The starting point for an objects-first approach is the definition of a class. A class is defined as a series of properties that reflect the state of an object and a series of capabilities that describe the appropriate actions for an object [11]. Using this definition, the objects-first approach stresses the differentiation between the class and the object. The student learns that a class refers to the "blueprint" or description of the properties and capabilities of an object and the object itself represents a discrete instance of the class definition. At this point, the instructor and student can engage in discussions that explore encapsulation as one of the three pillars of object-oriented programming.

Once students are able to differentiate between classes and objects, the objects-first approach examines relationships between classes. Students learn how to define and differentiate between composition (the has-a relationship) and association (the knows-a relationship) [2,3,11]. In composition, the construction of a single object invokes the construction of additional objects. The combination of the original object along with the additional objects completely describes the "composite" object. In association, a reference to one object is passed to another object during its construction. Therefore, unlike composition, association is not responsible for the creation of the related object.

After composition and association are covered, students learn about inheritance and polymorphism as key principles [2,3,11]. When studying the concept of inheritance, students learn how to extend classes, adding specificity to generalized classes. For polymorphism, students learn that for a generalized method invocation, a class can "do what it is supposed to do" by nature of the object-oriented paradigm.

The primary concepts of the objects-first approach sometimes prove difficult for students to grasp. To help provide structure, programming exercises are used to ground the concepts. Unfortunately, early attempts at text-based examples to motivate exploration of class relationships were not well received by students. In examining our student base, it was clear that for students who define programming though video games and other media-rich experiences, the concept of text-based computing interactions would not prove compelling. To counter this problem, two of the authors along with other faculty adapted objects-first materials from Brown University [6] in order to provide students with graphical introductory programming exercises. Instead of text-based experiences, students created programs designed to illustrate principles of composition through image, interactivity and sound. For composition, students created a breakfast table that had various breakfast items and an animated house that was decorated with a number of interactive knickknacks. For association, students created graphical picture books in which association allowed for interactive control of image and sound. Such exercises proved encouraging and successful in their own right [11]. However, the authors believe that a game-based three-dimensional collaborative virtual environment may prove to both more motivating and more accessible to students.

3. A Virtual Environment for Exploring Objects-First Concepts

At RIT, we have been working to create a collaborative virtual environment designed for the express purpose of teaching introductory programming. The Multi-User Programming Pedagogy for Enhancing Traditional Study (M.U.P.P.E.T.S.) [4,9] system directly challenges assumptions that the approach to learning programming must be theoretical and abstract. Instead, the environment is designed to allow students to explore programming through experimentation and careful observation of cause and effect. The M.U.P.P.E.T.S. environment speaks to students as it provides a framework in which students can create interactive 3D content in a game-like virtual world. This provides for an introductory programming experience that is much more akin to programs the students have used themselves. Additionally, the system seeks to create a stronger bond between upper and lower division students by allowing them to co-inhabit our virtual space.

Figure 1: A complex M.U.P.P.E.T.S. scene with several graphics effects.

One of the goals of the M.U.P.P.E.T.S. environment is to provide the student with the ability to create and add content to the virtual space with ease. Extending the *MuppetsObject* base class, which amounts to a few lines of code, does this for the student. Students create a simple scene of primitives in their very first experience with the system. These scenes, while rudimentary, still exhibit all of the graphical niceties found in modern commercial games: glow, reflection, refraction, shadows, fog, smoke, as well as other graphical effects. An example is shown in Figure 1, in which the student is making use of several advanced effects such as glow and transparency. The purpose here is not to add complexity, but rather to convince the first time programmer that they are working with a professional medium, which has been a criticism of systems that have tried this in the past. Along that same vein, the programming language of M.U.P.P.E.T.S. is not a "toy" language – the system uses standard Java from Sun Microsystems, and all features of the language are available. In order to achieve this, the system is architected as merely a layer over the underlying compiler. Support for Microsoft's C# programming language is now in development, through a similar mechanism.

The M.U.P.P.E.T.S. system is divided into three major components: the virtual world, the command console, and the integrated development environment (IDE). Each of these modes can be enabled from the others – there is no need to exit the world to test a possible solution to a code problem. The virtual world is a standard gaming environment in which a user has an avatar of self-representation, and views the world from either a first- or third-person perspective. Movement is accomplished through the 'mouse look and key move' metaphor common to games of the current generation.

The command console is a single pane from which new objects can be instantiated, and console variables set that control the various options available to the system (whether or not to use shadows, the view distance, or other features).

The final mode available within the system is the IDE. The IDE specifically contains features that students felt were critical to a professional programming environment: line numbering, color-coded syntax highlighting, context-sensitive auto-complete, as well as other features associated with available IDE implementations. The IDE layers over the existing world when desired. Anecdotal student feedback has consistently pointed out that the ability to code, compile, create, destroy and re-code without leaving the world has been a very desirable feature of the system.

4. Motivating Fundamental Object-Oriented Concepts

In this section, we demonstrate how M.U.P.P.E.T.S. can be used to motivate critical concepts in object-oriented design and implementation, including differentiation of class and object, inheritance, composition, and association.

4.1 Class vs. Object and a First Look at Inheritance

When students are initially introduced to M.U.P.P.E.T.S., one of the first things they wish to do is to create a simple object within the virtual world. To demonstrate how this is done, the instructors present a program, which creates a simple object with the least lines of M.U.P.P.E.T.S. code. In M.U.P.P.E.T.S., the simplest code allows for the construction of a primitive shape.

```
// ETCube.java

import muppets.core.*;

public class ETCube extends MuppetsObject {

   public ETCube() {
      super();
      this.setPrimitive(CUBE);
   }
}
```

Figure 2: *ETCube* – A simple M.U.P.P.E.T.S. class

As seen in Figure 2, the *ETCube* class has many of the key features one would expect in an object-oriented class definition. First, the class definition allows for the discussion of import statements, comments, constructors, and method invocation. However, more importantly, it provides the student with his or her first exposure to the concept of inheritance. Even in the simplest M.U.P.P.E.T.S. program, all entities within the world must extend the functionality of the *MuppetsObject* class. This immediately allows the instructor to challenge students to ponder the concept of inheritance and leads to discussion regarding traversing the hierarchy for method invocation as well as the use of the keyword *super* in code. The example also allows students to experiment with keywords such as *this*.

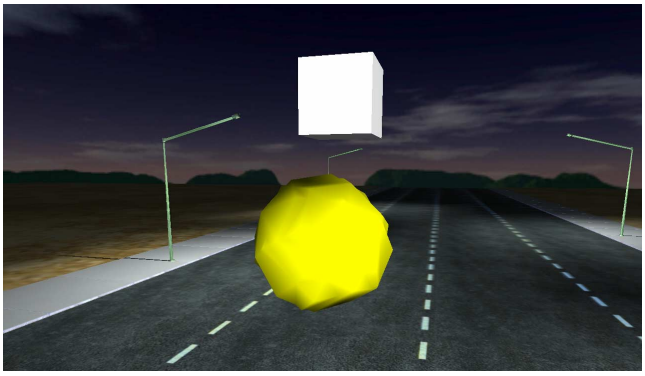

Figure 3: Creating an Instance of an *ETCube*

Once a student successfully creates and compiles the *ETCube* code file, he or she can use the *ETCube* definition within the M.U.P.P.E.T.S. world. As shown in Figure 3, students can create new instances of the *ETCube* wherever they wish, merely by navigating to a new location (using the yellow spherical avatar) in the virtual world, invoking the console, and typing *new ETCube*. Students can determine if their creations work satisfactory within a local "sandbox" area, and when they are satisfied with their creations, they can present their results to their instructor or peers.

In many ways, the *ETCube* example is similar to the ever-familiar *HelloWorld* example in that it represents the minimum functionality necessary to create a functional program in the representative language. However, unlike *HelloWorld*, it promotes strong object-oriented practices from the first exposure with the language. Furthermore, the *ETCube* example combined with the M.U.P.P.E.T.S. environment provides visual, interactive feedback to the student.

4.2 A Deeper Look at Inheritance

As students explore object-oriented issues, they begin to learn that not only can they extend built-in objects such as *MuppetsObject*, but also that they can use inheritance to make their own creations more powerful.

```
// ETColorBox.java

import muppets.core.*;

public class ETColorBox extends MuppetsObject {

   private static final float SIZE_LENGTH = 7.0f;
   private static final float SIZE_WIDTH = 4.0f;
   private static final float SIZE_HEIGHT = 3.0f;
   private static final float COLOR_R = 1.0f;
   private static final float COLOR_G = 0.0f;
   private static final float COLOR_B = 0.0f;

   public ETColorBox() {
      super();
      this.setPrimitive(CUBE);
      this.setScale(SIZE_LENGTH,SIZE_WIDTH,
                    SIZE_HEIGHT);
      this.setColor(COLOR_R, COLOR_G, COLOR_B);
   }
}
```

Figure 4: The *ETColorBox* class

To demonstrate, students start out with a rather simple class of their own construction, such as *ETColorBox* in Figure 4. This class creates a box, consisting of a fixed size and a fixed color, in this case, red. In classroom discussion, students are challenged by the instructor to think about how additional functionality, such as animation, could be added to the class. There are, of course, many different thoughts regarding how this can be accomplished. Some students suggest that the animation code should be added directly to the existing class. The opposing view argues that the original functionality of the class should remain intact, just in case we need to use *ETColorBox* in its original context.

```
// ETRotateBox.java

import muppets.core.*;

public class ETRotateBox extends ETColorBox {

   private static final float RECT_ROTATE = 1.0f;

   public ETRotateBox() {
      super();
   }

   public void update(float aDeltaTime) {
        super.update(aDeltaTime);
        this.rotate(RECT_ROTATE,1.0f,0.0f,0.0f);
   }
}
```

Figure 5: *ETRotateBox* extends the *ETColorBox* class

Eventually, this discussion proves to be a catalyst for the discussion of inheritance within one's own work. Using a UML class diagram, design of the *ETRotateBox* class is discussed, which inherits functionality from *ETColorBox*. From this design, students proceed to an implementation of the *ETRotateBox* class, similar to the solution depicted in Figure 5. Within the M.U.P.P.E.T.S. environment, students can create both instances of *ETColorBox* as well as *ETRotateBox* in order to observe the differences between the original class and the extended class.

4.3 Composition

The next step for students is to understand the concept of composition. Composition provides for the *has-a* relationship and implies that the constructor of a primary object has the responsibility for creating instances of sub-objects that comprise the composite object.

One method that is used for demonstrating composition to students is the creation of a face in the M.U.P.P.E.T.S. environment. The in-class conversation starts with students discussing the components that comprise a face. Immediately, students mention critical components such as eyes, noses, mouths, ears, and other features. However, upon closer inspection, students realize that many of the components themselves are constructed of smaller parts. For example, the eyes can be thought of as consisting of the sclera, iris, and pupil. The instructor uses the various levels of detail to start the design process.

To begin, the instructor starts to discuss the design of an eye. The students work together to design the UML class diagram that represents the eye, as seen in Figure 6.

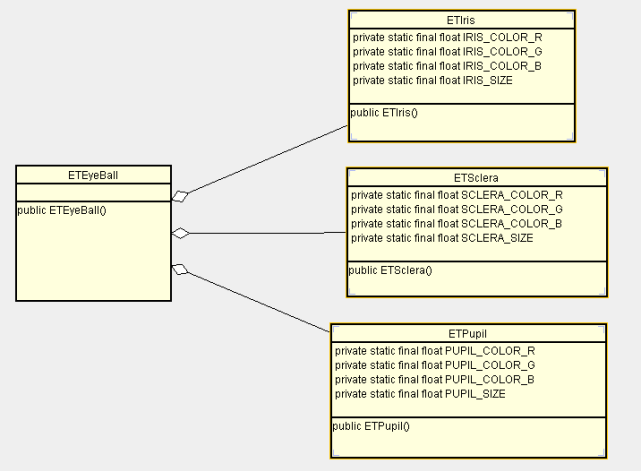

Figure 6: *ETEyeBall* UML class diagram

The principle of composition is discussed in relationship to the class diagram. From here, students start working on the individual components of the eye. If they work in groups, each student codes one part of the eye. For example, Figure 7 illustrates the code needed to create the Sclera.

```
// ETSclera.java

import muppets.core.*;

public class ETSclera extends MuppetsObject {

   private static final float SCLERA_CR = 0.95f;
   private static final float SCLERA_CG = 0.95f;
   private static final float SCLERA_CB = 0.95f;
   private static final float SCLERA_SIZE = 0.5f;

   public ETSclera() {
      super();
      this.setPrimitive(SPHERE);
      this.setScale(SCLERA_SIZE, SCLERA_SIZE,
                    SCLERA_SIZE);
      this.setColor(SCLERA_CR, SCLERA_CG,
                    SCLERA_CB);
   }
}
```

Figure 7: ETSclera Code

Once students complete their individual eye parts, the code that performs composition is created. Figure 8 illustrates this code. From here, students can create floating eyeballs within their world as a means of determining that composition worked!

Once students understand the construction of a single eyeball using composition, they use the same principle to build the entire face. As part of the continued exercise, they learn about code reuse (since they use the eyeball they created earlier) and they receive additional reinforcement regarding the composition process.

```
// ETEyeBall class

import muppets.core.*;

public class ETEyeBall extends MuppetsObject {

   private ETSclera _sclera;
   private ETIris   _iris;
   private ETPupil  _pupil;

   public ETEyeBall() {
      super();
      this.setPrimitive(NODRAW);

      _sclera = new ETSclera();
      this.addChild(_sclera);

      _iris = new ETIris();
      _iris.move(0.0f, 0.0f, 0.051f, true);
      this.addChild(_iris);

      _pupil = new ETPupil();
      _pupil.move(0.0f, 0.0f, 0.075f, true);
      this.addChild(_pupil);
   }
}
```

Figure 8: ETEyeBall Code

Students are encouraged to customize their faces so that each student's work is different. Figure 9 demonstrates a completed face within the M.U.P.P.E.T.S. environment.

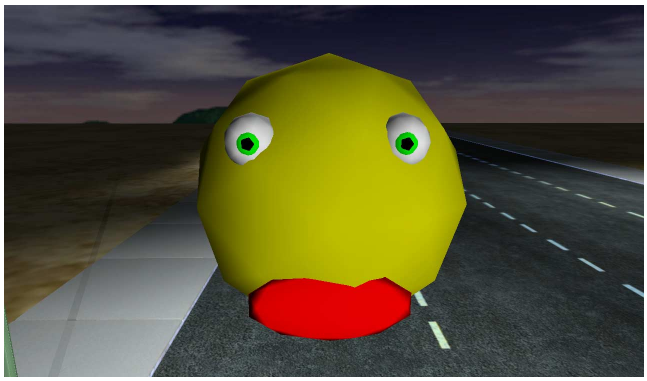

Figure 9: Instance of an ETFace object in the M.U.P.P.E.T.S. world

4.4 Association

Once students build faces using composition, the concept of association is explored through the construction of a trigger that affects a change upon the face. In this case, an animated cube within the M.U.P.P.E.T.S. world acts as a trigger to change the face from yellow to blue. As seen in Figure 10, the instance of the face is passed to the trigger constructor such that a *knows-a* relationship can be established. When the student constructs the scene, he or she can navigate the avatar to the trigger point and activate the trigger to affect the change. The students are challenged to discuss why the relationship between the face and the trigger is best expressed with association rather than composition.

```
// ETScene.java

import muppets.core.*;

public class ETScene extends MuppetsObject {

   private ETFace     _face;
   private ETTrigger  _trigger;

   public ETScene() {
      super();
      this.setPrimitive(NODRAW);

      _face = new ETFace();
      this.addChild(_face);

      _trigger = new ETTrigger(_face);
      this.birth(_trigger);
   }
}
```

Figure 10: Example of association within ETScene

5. Future Work

It is the intent of the authors to continue to explore ways of creating objects-first programming exercises that are appropriate for demonstration and student critical thinking within collaborative virtual environments. The authors plan to test their approach multi-institutionally and to analyze the effect of using such assignments upon learning and comprehension within the introductory course.

6. Acknowledgements

The authors would like to thank RIT's Provost's Learning Initiatives Grants program, the RIT Information Technology Department, and Sun Microsystems for providing initial funding and support for the M.U.P.P.E.T.S. project. Furthermore, the authors wish to acknowledge Microsoft Research, and the MSR External Research and Programs Group for continued research funding of the M.U.P.P.E.T.S. project. The authors would also like to thank all past and present students who have contributed to the M.U.P.P.E.T.S. project

7. Further Information

Additional information regarding the M.U.P.P.E.T.S. project can be found at http://muppets.rit.edu.

8. References

[1] ACM/IEEE Joint Task Force on Computing Curricula. *Computing Curricula 2001: Computer Science*, December, 2001

[2] Alphonce, C. and Ventura, P. Object Orientation in CS1-CS2 by Design, *In Proc. of the 7th Annual ITiCSE Conference*, Aarhus, Denmark, 2002,70-74

[3] Alphonce C. and Ventura, P. Using Graphics to Support Teaching of Fundamental Object Oriented Principles, *In OOPSLA 2003 Educator's Consortium Companion*, 2003, 156-161

[4] Bierre, K. and Phelps, A. The Use of M.U.P.P.E.T.S. in an Introductory Java Programming Course, *In Proc. of the 5th Annual SIGITE Conference*, 2004, 122-127.

[5] de Raadt, M., Watson, R., and Toleman, M. Introductory Programming: What's Happening Today and Will There Be

Any Students to Teach Tomorrow?*, Proc. of the 6th ACE Conference*, Dunedin, New Zealand, 2004, 277-282

[6] Duvall, R., Chotin, M., Neuringer, M., Goldberg, D., and d van Dam, A.. Object-Oriented Programming Chapters (draft): Online: http://www.cs.brown.edu/courses/cs015/2001/Chapters/contents.html

[7] Guzdial, M. and Soloway, E. Log on Education: Teaching the Nintendo Generation how to Program, *Communications of the ACM,* 45(4), 2002

[8] Kölling, M. and Rosenberg, J. Guidelines for Teaching Object Orientation with Java, *In Proc. of the 6th Annual ITiCSE Conference, Canterbury, UK, 2001, 33-36*

[9] Phelps, A., Bierre, K., and Parks, D. M.U.P.P.E.T.S: Multi-user Programming Pedagogy for Enhancing Traditional Study, *In Proc. of the 4th CITC Conference,* 2003, 100-105

[10] Sun Microsystems. Java Technology Home Page: Online: http://www.javasoft.com

[11] Ventura, P. On the Origins of Programmers: Identifying Predictors of Success for an Objects First CS1. Ph.D. Thesis, University at Buffalo, Buffalo, NY, 2003.