A Graphics Package for the First Day and Beyond

Michael H. Goldwasser
Dept. Mathematics and Computer Science
Saint Louis University
221 North Grand Blvd
St. Louis, Missouri 63103-2007
goldwamh@slu.edu

David Letscher
Dept. Mathematics and Computer Science
Saint Louis University
221 North Grand Blvd
St. Louis, Missouri 63103-2007
letscher@slu.edu

ABSTRACT

We describe `cs1graphics`, a new Python drawing package designed with pedagogy in mind. The package is simple enough that students can sit down and make use of it from the first day of an introductory class. Yet it provides seamless support for intermediate and advanced lessons as students progress. In this paper, we discuss its versatility in the context of an introductory course. The package is available at `www.cs1graphics.org`.

Categories and Subject Descriptors

I.3.4 [**Computer Graphics**]: Graphics Utilities—*paint systems*; K.3.2 [**Computers and Education**]: Computer and Information Science Education—*computer science education*; D.1.5 [**Programming Techniques**]: Object-oriented Programming

General Terms

Design, Human Factors

Keywords

CS1, Python

1. INTRODUCTION

Computer graphics has served as a domain for teaching computer programming dating back to Logo's turtle graphics in the 1970s [1, 8]. Manipulating graphics can be both rewarding and motivating, providing tangible feedback for students. Most modern languages support industrial graphics libraries, but these are widely regarded as unsuitable for beginning programmers. This has led to decades filled with custom graphics packages for use in education. In Java, the most widely used packages today include Java Power Tools [9], objectdraw[2], and `acm.graphics` from the ACM Java Task Force [10]. Guzdial's multimedia computation package can be used with Java or Jython [7], and Zelle provides an object-oriented drawing package for Python [11].

SIGCSE'09, March 3–7, 2009, Chattanooga, Tennessee, USA.

In 2005, we redesigned our object-oriented CS1 course using Python as the instructional language [6]. We wished to include the manipulation of 2D graphical objects, yet none of the existing Python packages satisfied our requirements. Faced with creating a package from scratch, we took the opportunity to develop a design that graciously combined our favorite features of existing (Python and Java) libraries.

Our fundamental goal was to achieve a design that is simple enough to use from the very first day, yet rich enough to provide a scaffold for more advanced lessons. Our criteria for simplicity is that a student can sit down and create interesting images, with only the most modest preliminary coaching. The intermediate and advanced lessons enabled by our graphics package involve flow of control, container classes, object-oriented principles, inheritance, recursion, and event-driven programming. Yet we insist that the support for these lessons not adversely impact the ease of use for that first day.

With this paper, we wish to announce the availability of the package and to describe its support of pedagogy. The package is an object-oriented Python module named `cs1graphics`, available at `www.cs1graphics.org`. We believe that its combination of simplicity and functionality makes it a valuable tool for students in a CS0 or CS1 course, for high school students, and even for an enthusiastic ten-year old.

2. EXISTING GRAPHICS PACKAGES

Although the use of Python in education has increased in recent years, the available pedagogical graphics packages are not nearly as mature as those for Java. The standard industrial package in Python is Tkinter, which is a wrapper for the Tcl/Tk widget set. Yet it relies upon a complex set of classes, requires keyword parameter passing, and draws attention to the event loop even when creating static images. The features that make Tkinter a useful tool for experienced programmers make it unsuitable for beginning students.

For educators, the most widely used Python graphics package is that of Zelle [11]. It performs quite well as that "first day" package, allowing students to create a graphics windows and to draw basic geometric shapes. Yet Zelle's package lacks the more intermediate and advanced support, as it does not allow existing shapes to be resized, rotated, or flipped, does not support any composite collection abstraction, nor is it easily extensible through inheritance.

Guzdial's Jython/Java library [7] is a wonderful package for manipulating multimedia formats. However, it does not have adequate support for creating and manipulating images composed from geometric objects.

3. THE FIRST DAY

An introduction to cs1graphics begins with the Canvas class, representing a window upon which we can draw. The mutable attributes of a canvas include its height, width, background color, and title. Using Python's interpreter, students can immediately begin experimenting as follows.

```
>>> from cs1graphics import *
>>> paper = Canvas( )
>>> paper.setBackgroundColor('skyBlue')
>>> paper.setWidth(400)
>>> paper.setHeight(300)
>>> paper.setTitle('My World')
```

Each command is executed immediately when entered, with the tangible effect visible to the student. So after instantiating the original canvas, a window immediately pops up on the screen. After the call to setBackgroundColor, the color of the canvas window changes. Such isolated manipulation of the canvas serves as a warmup; the next stage is for students to place objects on the canvas.

The core of our module is a hierarchy of Drawable objects. Students can be led through the creation of shapes and the placement of those shapes on a canvas. Continuing our earlier interpreter session, the syntax is as follows.

```
>>> sun = Circle( )
>>> sun.setRadius(30)
>>> sun.setFillColor('yellow')
>>> sun.move(250, 50)
>>> paper.add(sun)
```

The initial construction and configuration of the circle have no immediate effect on the canvas. It brings that shape into existence, but it will not be visible until explicitly added to the canvas. We could have added the circle to the canvas before configuring its properties, in which case each intermediate state would be rendered.

Other classes of drawable objects include Rectangle, Square, Path (i.e., a polyline), Polygon (i.e., a closed and fillable path), Text, and Image (wrapper for standard image files).

We describe our system to students using the following physical analogy. A canvas is like a bulletin board and the shapes like paper cutouts that can be fastened to the canvas using a thumbtack. Each shape has a local reference point that corresponds to the spot through which the thumbtack is placed. The position of the shape relative to the canvas is described using the coordinate system of the canvas.

After walking out of the classroom on this first day, a student can compose static scenes. With the addition of the sleep command, straight-line code can be used to develop dynamic animations. The first take-home assignment in our course asks students to create a multiframe animation of their favorite animals. Students have a great time with this beginning, often submitting assignments with hundreds of lines of code and great artistic details. In the context of a CS0 course, such a project might serve as the pinnacle of a brief programming unit.

4. INTERMEDIATE LESSONS

After completing the first assignment, our CS1 students begin to understand some important programming concepts. They also realize that there must be more convenient ways to perform some tasks from their original animations.

Control Structures. Even with straight-line code, the use of the sleep command provides an aspect of timing that reinforces the concept of flow of control. In initial projects, students likely relied on a "copy-and-paste" style, perhaps to animate motion by a sequence of move, sleep, move, sleep, move, sleep commands. Even a novice student will realize that there must be a better way. This provides a natural segue to discussing control structures, such as the use of a loop to express the repetition of an object sliding across the canvas. Additional structures can be motivated in projects, such as a conditional for simulating the bouncing of a ball against the wall, or a combination of loops and conditionals for drawing the alternating colors of a checkerboard.

Principles of Computer Graphics. To support the first day experience, our software intentionally renders the effect of each statement as it is executed. Yet for more complex scenes, this can result in undesired visual artifacts. If dozens of individual manipulations are required to compose a new scene, the viewer might notice intermediate frames of the incomplete image. Students will naturally want to know how they could avoid such problems; the technical solution that is desired is to double-buffer images.

Our library supports this by having automatic refreshing of the canvas as the default for beginners, but a method to toggle the refresh model. When automatic refreshing is off, the programmer must explicitly call a refresh method to force an update of the canvas. This teaches an important technique in computer graphics while also pushing students to recognize the distinction between the internal state of the model and the currently visible image.

Our software also supports general transformations for all drawable objects through methods such as scale(factor), rotate(angle), and flip(reflectionAngle). This functionality is useful both in practice and for pedagogy. When scaling or rotating an object, the action must be performed relative to some fixed point. Recall that we earlier introduced the concept of a reference point as the hypothetical point at which an object is tacked to the canvas. A default reference point is defined naturally for each class, yet the user can reposition an object's reference point using a method adjustReference(dx, dy). With our thumbtack analogy, the shape is not moved at all relative to the canvas, rather the thumbtack holding it is taken out, repositioned, and reinserted. As such, this call has no immediately visible effect. But it can be a precursor to a rotation or scaling to achieve desired results. Figure 1 shows two possible rotations of a square, depending upon the placement of the reference point. Technically, the reference point need not overlap the area of the shape; for example we might place a moon's reference point aligned with a planet to simulate an orbit.

Figure 1: Rotating a square about two different reference points. On the left, the square is rotated about its center point. On the right, the same square is rotated about its lower-left corner.

Composition. In the first assignment, a student may have "moved" a dog by first moving the head, then the body, followed by the tail. If an animal has many individual components, this approach may produce visual artifacts, such as an instantaneous view of the dog's head detached from the rest of the body. Even though double buffering might be used to remedy the artifact, we prefer that the "dog" be modeled as a single unit rather than as independent components. With the piecewise design, a conceptually simple manipulation of the dog may require dozens of lines of code.

To support a better design our library defines a Layer class that is a composite, serving as both a container and a drawable object. We describe it by physical analogy as a very thin clear film akin to the animation cells of the early 1900s. We can attach other shapes directly to this film rather than to the canvas. The layer can then be placed upon a canvas. As a composite, a layer allows a programmer to naturally group shapes that are conceptually related. Furthermore, since layers are themselves Drawable objects, they can be moved, scaled, flipped, or rotated coherently and without temporal artifacts. This provides a means for reinforcing good organization and design. Also, the functionality is quite powerful; it can be challenging to accurately scale or rotate a composite figure when modeled as independent shapes.

The use of layers requires awareness of distinct frames of reference and contexts. The layer has its own coordinate system, just as a canvas has a coordinate system. When an object is added to a layer, its position is specified relative to the origin of the layer (which need not coincide with the origin of the canvas). Relative depths are resolved only within the context of the immediate container. That is, the rendering order for objects on a layer is based upon the depths of those objects. But the layer is a single composite object; how that layer should be placed relative to other objects on a canvas is affected by the layer's depth attribute. We note that a layer can be added as a component onto another layer, allowing for greater levels of abstraction and an exploration of recursive structures. The following code demonstrates the use of a layer to model a simple car.

```
car = Layer( )
tire1 = Circle(10, Point(−20,−10))
tire1.setFillColor('black')
car.add(tire1)

tire2 = Circle(10, Point(20,−10))
tire2.setFillColor('black')
car.add(tire2)

body = Rectangle(70, 30, Point(0, −25))
body.setFillColor('blue')
body.setDepth(60)              # behind the tires
car.add(body)                  # (as default depth is 50)

paper.add(car)
car.move(110,180)              # move the car as a whole
car.scale(2.0)                 # scale the car as a whole
body.setFillColor('purple')    # mutate component
```

Three individual components are added to the layer instance and then the layer is added to the canvas. Notice that the layer can be moved or scaled as a whole, yet that individual components from that layer (e.g., the body) can still be identified and manipulated directly. The depth attribute for the three components affects the relative placement of those components within the layer. The car's depth relative to the canvas would be controlled independently. This allows us to incorporate the car into an existing scene, such as the one shown in Figure 2.

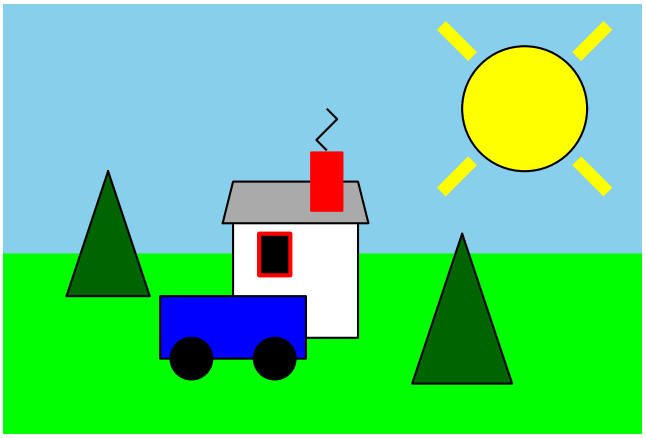

Figure 2: An example of a car incorporated into a scene.

5. BASICS OF OBJECT-ORIENTATION

The design of cs1graphics is heavily object-oriented, but it is up to the instructor whether or not to draw attention to this fact. In this section, we highlight several potential lessons about object-oriented principles.

Calling Syntax. With many objects in play, the significance of the identifier in the method-calling syntax can be emphasized. That is, a programmer cannot issue the unqualified command move(110,180), but instead must explicitly identify the object to manipulate as in car.move(110,180).

Object State. The basic manipulations of a canvas and the drawable objects instills an awareness of objects. Students see that the state of an object can be mutated, and that the state of one instance is independent of other instances. They will naturally observe that different classes support a different menu of behaviors (e.g.,that Canvas instances support setHeight while Circle instances support setFillColor).

Type Awareness. Although Python is dynamically typed, our functions explicitly error-check all parameters, raising meaningful errors when appropriate. Our classes and methods are also well documented with Python docstrings, available to students through the interpreter's help command.

Inheritance Hierarchy. The students' use of inheritance in their own code will be explored as an advanced lesson. Long before that, we highlight our own use of inheritance in defining the Drawable hierarchy. The commonality among classes' behaviors makes it relatively easy to learn to use additional classes; only those methods specific to each new class need to be mastered. Early exposure to such a hierarchy can strengthen students' later ability to identify similarities and build abstractions in their own designs.

Polymorphism. Our package demonstrates several forms of polymorphism. As a classic example, Drawable types each support a method named _draw yet with different underlying implementations.

As an example of parametric polymorphism, whenever a color is to be specified as a parameter, that color can be designated as a string from a predefined palette of names, as an (r,g,b) tuple, or an existing instance of the Color class. We determine the representation type at runtime. Many of our functions use optional parameters to provide multiple calling signatures. For example, our early code fragments have demonstrated the constructor signature Circle( ) as well as Circle(10, Point(20, −10)).

6. INHERITANCE

In the preceding section, we noted that the hierarchical design of the package provides students initial exposure to the concept of inheritance. When it comes time to discuss the use of inheritance for *user-defined* classes, we leverage our package in several ways.

Examples of Single and Multiple Inheritance. As a precursor to having students use inheritance in their own classes, we offer a behind-the-scene look at our own source code to demonstrate the inheritance mechanisms. For example, we can show how the FillableShape class specializes the Shape class or how the Square class specializes the Rectangle class.

We can also motivate and demonstrate interesting examples of multiple inheritance in the design of our our package. We already discussed the Layer class as a hybrid. By inheriting from Drawable, it receives attributes such as a depth and a reference point, and behaviors such as scale and rotate. At the same time, both canvases and layers serve as containers for drawable objects. We define an underlying _GraphicsContainer class that serves as a parent to both Canvas and Layer, providing the ability to add and remove objects and to compute the proper painter's ordering. Based on these classes, our Layer implementation uses multiple inheritance, declared as Layer(Drawable, _GraphicsContainer).

Extensibility. After demonstrating the technique and syntax of inheritance with our own examples, we ask students to use the technique. Our hierarchy of Drawable objects is intentionally extensible, with several good entry points for inheritance (in fact, quite similar to `acm.graphics` [10]).

One possible usage is to inherit from one of our most-derived classes. For example, a Star class can be defined as a specialization of the existing Polygon class, with a constructor that configures the initial placements of the points. The star inherits all other behaviors from the parent class, including being fillable, movable, and rotatable. Of course, an important lesson is that *all* methods of the parent class are inherited by default, and so a user could invoke the addPoint method upon one of our star instances. Robustness could be added by overriding that method to **pass** or by raising an exception.

A more general entry point for inheritance is the Layer class. A child class can assemble components internally, while inheriting behaviors such as move, scale, and rotate. It is also possible to inherit directly from the abstract Drawable class by providing an implementation of a _draw method to compose a rendered image from underlying components.

Revisiting the First Assignment. The extensibility of our inheritance hierarchy provides an opportunity to improve the design of the students' original animation project. The animals were first implemented as a set of independent parts. The design was improved with the subsequent introduction of the Layer composite, but that approach is still less than ideal. The Layer class supports general behaviors like rotate and scale, but not behaviors specific to the context of an animal (e.g., wag a tail). Also, the layer-based approach does not allow for the animal to be reused in a different project without significant duplication of code.

Therefore, we ask students to revisit their earlier work and to design and implement a specialized class to represent their animal. Past students have designed a Horse that rears up on its hind legs, a Bird that flaps its wings and flies from point to point, and a Dog that sits, wags its tails, and barks. This exercise provides students the opportunity to use creativity as a designer. They envision interesting behaviors, select appropriate signatures for the methods, and choose the internal representation for an animal's state.

7. RECURSION

Recursion can be a challenging topic for many students to grasp. Bruce *et al.* advocate for the early introduction of structural recursion [3]. They provide examples such as a ringed target or a fractal-based broccoli rendered using their `objectdraw` graphics package. While structural recursion can be demonstrated with non-graphical examples [5], the graphical examples carry great intuition.

Within cs1graphics, recursive classes can be developed by directly extending our Drawable class, or by making use of Layer composites which can be nested arbitrarily. As an example, a typical Bullseye instance can be modeled as an outer circle with a smaller inner Bullseye instance centered on top, as diagrammed in Figure 3. Here is the constructor for a Bullseye class that extends the Drawable class.

```
def __init__(self, bands, radius, colorA, colorB):
  Drawable.__init__(self)            # parent constructor
  self.outer = Circle(radius)
  self.outer.setFillColor(colorA)

  if bands > 1:          # construct inner bullseye recursively
    newRad = radius * (bands - 1) / bands
    self.inner = Bullseye(bands-1, newRad, colorB, colorA)
  else:
    self.inner = None
```

Our instance has two data members: outer represents the outer circle and inner represents a bullseye instance with one less band, smaller radius, and inverted colors. As a base case, a bullseye with one band has an outer circle but no inner component.

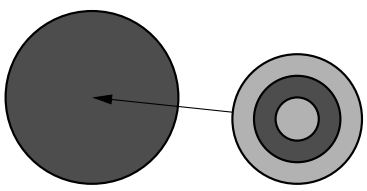

Figure 3: Recursive modeling of a Bullseye.

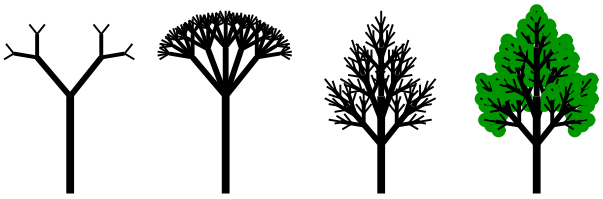

Figure 4: Instances of a Tree class.

Methods such as _draw follow a simple pattern, acting upon the outer circle followed by the inner bullseye, if any. A similar approach can be used to model other examples of structural recursion, such as the trees shown in Figure 4.

8. EVENTS AND USER INTERACTION

Bruce *et al.* also demonstrate the use of event-driven programming in teaching introductory topics [4]. To facilitate such an approach, our library supports two different models for dealing with events.

Appropriate for those early days is a basic single-threaded waiting model. The canvas and all drawable shapes support a wait( ) method that causes execution of the calling thread to be suspended until an event such as a mouse click is received by the indicated object. The wait function returns an Event instance that stores information about the event that occurred. As a simple example, here is a program that repeatedly draws a red ball centered on the point at which the user clicks on a canvas.

```
paper = Canvas( )
while True:
  event = paper.wait( )      # wait indefinitely for user
  ball = Circle(10, event.getMouseLocation( ))
  ball.setFillColor('red')
  paper.add(ball)
```

For a more advanced treatment of event-driven programming, the library supports a listener model for events. An EventHandler serves as a base class, allowing handlers to respond to selected event types by overriding the handle method. As a simple example, here is code using the listener model to provide functionality similar to the preceding example.

```
class CircleDrawHandler(EventHandler):
  def handle(self, event):
    if event.getDescription( ) == 'mouse click':
      ball = Circle(10, event.getMouseLocation( ))
      ball.setFillColor('red')
      event.getTrigger( ).add(ball)

paper = Canvas( )
paper.addHandler(CircleDrawHandler( ))
```

Yet the listener model is far more general, as events can be simultaneously monitored by a combination of listeners. In the simpler wait model, the flow of control must be blocked waiting on a specific object.

Our library also supports text boxes, buttons, and a few other basic GUI widgets as a preview of what students would see in an industrial strength library for graphical interfaces.

9. CONCLUSIONS

With this paper we introduce cs1graphics, a new Python drawing package designed with pedagogy in mind. The strength of our design is the combination of simplicity and functionality, providing support for beginning and advanced lessons in the context of a CS1 course.

Comparing our package to existing pedagogical packages, we find that others are either too limited or too complex. For example, Zelle's Python package [11] is sufficiently simple for beginners, but lacks more advanced features (e.g., general transformations, composition, an extensible hierarchy). At the other extreme, the `acm.graphics` library in Java [10] has similar functionality to our library, yet appears less approachable on the first day without scaffolding.

10. REFERENCES

[1] H. Abelson and A. diSessa. *Turtle Geometry*. The MIT Press, 1980.

[2] K. B. Bruce, A. Danyluk, and T. Murtaugh. A library to support a graphics-based object-first approach to CS 1. In *Proc. 32nd SIGCSE Technical Symp. on Computer Science Education (SIGCSE)*, pages 6–10, Charlotte, North Carolina, Feb. 2001.

[3] K. B. Bruce, A. Danyluk, and T. Murtaugh. Why structural recursion should be taught before arrays in CS1. In *Proc. 36th SIGCSE Technical Symp. on Computer Science Education (SIGCSE)*, pages 246–250, St. Louis, Missouri, Feb. 2005.

[4] K. B. Bruce, A. Danyluk, and T. P. Murtaugh. Event-driven programming is simple enough for CS1. In *Proc. Sixth Annual Conf. on Innovation and Technology in Computer Science (ITiCSE)*, pages 1–4, Canterbury, United Kingdom, June 2001.

[5] M. H. Goldwasser and D. Letscher. Teaching strategies for reinforcing structural recursion with lists. In *Companion to 22nd ACM SIGPLAN Conf. on Object-Oriented Programming, Systems, Languages and Applications (OOPSLA)*, pages 889–896, Montreal, Quebec, Canada, Oct. 2007.

[6] M. H. Goldwasser and D. Letscher. *Object-Oriented Programming in Python*. Prentice Hall, 2008.

[7] M. Guzdial. *Introduction to Computing and Programming in Python: A Multimedia Approach*. Prentice Hall, 2005.

[8] H. Lieberman. The TV Turtle: a Logo graphics system for raster displays. In *The papers of the ACM Symposium on Graphic Languages*, pages 66–72, Florida, Apr. 1976.

[9] J. Raab, R. Rasala, and V. K. Proulx. Pedagogical power tools for teaching Java. In *Proc. Fifth Annual Conf. on Innovation and Technology in Computer Science (ITiCSE)*, pages 156–159, Helsinki, Finland, July 2000.

[10] E. Roberts, K. Bruce, R. Cutler, J. H. Cross II, S. Grissom, K. Klee, S. Rodger, F. Trees, I. Utting, and F. Yellin. The ACM Java task force: Final report. In *Proc. 37th SIGCSE Technical Symp. on Computer Science Education (SIGCSE)*, pages 131–132, Houston, Texas, Mar. 2006.

[11] J. M. Zelle. *Python Programming: An Introduction to Computer Science*. Franklin, Beedle & Associates, 2003.