Graphical API Documentation

The following document describes the graphical API framework, covering the general programming model, and then a description of the objects that the framework provides, and how they interrelate.

Programming Model

Upon program startup, a **world object** is instantiated to handle communications between the main program and the runtime system (in this case, the runtime system consists of both the application framework and the operating system). The world responds to events that the runtime system sends, and has control over the menu bar at the top of the window. The world also maintains three lists of objects, differentiated by type.

The first list contains objects having methods that are invoked by the world periodically at the rate of about every tenth of a second. These are known as **updateable objects**. When the world detects that about a tenth of a second has passed, the world calls the **update** method in each of the updateable objects in the list. Each object's update method can be different; one object might attempt to do animation, while another object may be checking to see if other objects happen to be near it at the time.

The second list contains objects having methods that are called whenever the screen needs to be redrawn or updated. These **drawable objects** have graphical representations associated with them. For example, a butterfly drawable object would have a graphical representation of a real butterfly associated with it. The graphical representations are composed of **drawing primitives**, such as circles, lines, and rectangles. These drawing primitives are stored in a drawing list within the drawable object. When the world realizes (either because the programmer asked the world to update the window, or because of a request by the runtime system) that it needs to redraw the screen, it calls the **draw** method in each of the drawable objects in the list. Each objects draw method will run through the object's drawing list and draw each of the primitives in the list.

The third list contains objects having methods that are called whenever the user attempts to interact with the program. These **interactable objects** have methods that are invoked depending on the way in which the user is interacting with the program, such as by clicking with the left mouse button or by moving the mouse around. When the world is notified by the runtime system that the user attempted to do something, the appropriate method in all the objects in the list is called. For example, if the user pressed down the left mouse button, then the method that handles left button down situations (**OnLButtonDown**) is called for all of the objects in the list. It is up to the individual method to decide whether or not it should do something about the event.

It should be noted that a single object could simultaneously be in more than one of these lists. It is quite possible for a butterfly object, for example, to be both drawable (displays a butterfly on the screen) and interactable (chases the user's mouse cursor).

The world is also in charge of the menu bar at the top of the window, including the save/load options. The world can load in objects from a text file and place them into the appropriate lists.

Programmers can augment this system by creating their own drawable, interactable, and updateable objects. This is accomplished by creating derived classes from the above classes, and overriding the appropriate methods with the programmers' own methods. Furthermore, when a new object is created, a world method, **addObject**, must be called to notify the world that a new object needs to be put into the above lists.

An Example Extension: The DraggableClass

This section describes how to go about creating an interesting drawable, interactable object. This tutorial should be read at the same time as you are perusing **draggableclass.cpp** and **draggableclass.h**, which are included in the framework distribution.

Figuring out the desired behaviours

The programmer should determine what she wishes the new class to do. Be very explicit about it, write it down, and then match the desired behaviours to the available methods/objects in the Object hierarchy to determine a point from which derivation is useful.

In this case, we wish to make a drawable object that can be dragged around by the user using standard click-and-then-drag style movement. The user must first press down the left mouse button, then move around the mouse while the left mouse button is down. When the user is satisfied by the object's new position, she needs to release (or depress) the left button. This has a number of implications:

- The draggable object must have both the qualities inherent in the drawable object and an interactable object. The draggable object must be a drawable object in order to be able to display itself on the screen. The draggable object must be an interactable object because it is responsive to user input. It turns out that the interactable object inherits the qualities of the drawable object, so we only need to derive the draggable object class from the interactable object class (see object.h for details).
- We wish to define our own behaviours when the user presses down the left mouse button, when moving the mouse around while the left mouse button is down, and when the user releases the left mouse button. Fortunately, the interactable object defines methods that handle those three situations. Our draggable object class must override these methods to do what we want.
- We wish our draggable object to be instantiable. It turns out that we will also need to add three methods to make this so: an update method, a print method, and a read method to our draggable object class. All three of these are pure virtual, so some kind of implementation is necessary to make our draggable object instantiable.

Given this information, then, it becomes much easier to see the structure of the DraggableObject as defined in draggableobject.h. For a fuller description of the methods, see the appendix in this document, as well as object.h in the distribution. In addition to the methods outlined above, a couple of extra member variables are necessary to perform the dragging motion, and a constructor is also defined to initialize them. We'll talk more about the extern const char* draggableObjectName in a moment.

Notice that each of the methods that we will override (except for the print and read methods) take in a pointer to a world. This is extremely important, as the world contains information about where the mouse actually is, and we will need to tell the world if the object has changed itself to the point where the window needs to be redrawn to take into account these changes.

Implementing the desired behaviours

Now, let's take a closer look at the implementation of the DraggableClass, in draggableclass.cpp, focusing on the OnLButton method. This method is invoked when the user presses down the left mouse button. However, as mentioned previously, the world calls the OnLButton method of all of the interactable objects when it receives notification from the runtime system that the user has pressed down the left mouse button. It does not matter that a particular interactable object is nowhere near where the user pressed the left mouse button; its OnLButton method will be called. This means that when a particular draggable object's OnLButton method gets called, we are not sure if the user is really trying to interact with that particular object.

So, that means that the first thing that needs to be done is to determine if the location of the mouse corresponds with the location of the object whose OnLButton method was called. The draggable class OnLButton method first asks the world for the current mouse coordinates, and determines if the user is clicking on the current object (we use the intersects method, detailed in the appendix and in object.h, to see if this is true). If the user isn't, then we exit the method with return value false, which tells the world that the event was not processed by this object. If the user is clicking on the object, then we perform some actions, and return true, which tells the world that this object has successfully processed the event.

All of the interactable object methods that we overrode are implemented in a similar way. First, these methods determine if they can handle the event. If they can, then do something. The return values of the other two methods are determined in the same way as the OnLButton method.

Of note is the usage of the world's **update** method in these methods. The update method is called when the object has changed to the point where the changes affect the graphical representation of objects on the screen. Note that the only time we make such a change is when we change the origin in the OnMouseMove method. It is generally a good idea, for performance reasons, to call the update method only when necesary.

Now, note the DraggableClass **update** method (this is not to be confused with the world's update method talked about earlier). It merely returns false (implying that nothing was done to modify the graphical output), since our desired behaviour enumerated earlier doesn't really require the use of the update method. However, it's implemented since the method is defined in a base class to be pure virtual, and we wish to instantiate objects of type DraggableClass.

Loading/Saving of DraggableClass objects

This is the final step necessary to create a DraggableClass that will work properly within this framework. All concrete instantiable classes that are updateable, drawable, or interactable must be able to read/write itself terms of the **object definition format**.

The object definition format

An object definition looks like:

```
class-identifier {
        attributes
}
```

The class-identifier, or header, is a unique name that can be recognized as belonging to a specific type of saved object. The attributes contain information that's specific to the object, and are enclosed in a pair of braces. Whitespace (carriage returns, tabs, etc) doesn't matter, so long as they exist between the class-identifier and the open brace, the brace and the beginning of the attributes, and the end of the attributes and the end brace. For example:

```
class-identifier { attributes }
```

is equivalent to the first object definition above, but is a little bit more compact.

The DraggableClass print method

The DraggableClass print method is used to save the world to a text file, and must be able to output information regarding the draggable class that we wish to save. We print out the class identifier string (which is defined at the top of the file), and then the opening brace. we decide that the only attributes that are interesting are the origin of the draggable object and the draw list (what the object currently looks like), and we can leverage off of the DrawableObject's print method to take care of these two attributes (see object.cpp for all the details). Finally, we print out the closing brace to complete the requirements for the object definition format defined above.

The DraggableClass read method

We must define a **read** method capable of reading in both an object definition as defined above and a variant of the object definition without the class identifier but with the braces and the attributes. So, not only must the read method handle input in the object definition format above, but must be able to handle input of the form

```
{
        attributes
}
```

where the classi-identifier string is missing.

Fortunately, this can be easily accomplished, by first reading in a single character. If the character is a '{', then we know that input will be in the truncated form, and for one reason or another the class-identifier string is missing. Otherwise, we should go ahead and read in the class-identifier, and then read in the '{' that proceeds it. Then, it's a simple matter or reading in the attributes (just as we leveraged off of the

DrawableObject's print method, we can leverage off of DrawableObject's read method to get the origin and the geometry), and finishing by reading in the end brace.

Modify the loadWorld function

The final task that we need to do is to modify the loadWorld function, located in worldio.cpp. The loadWorld function is called by the world when the user wishes to load in a world, which is accomplished by choosing the "load" menu option from the menu bar. This function reads in objects from the istream that's passed in via the following process:

Read in and set the world size
While there are more things to read in:
Read in the class-identifier
If the class identifier matches a known instantiable class then
Dynamicaly allocate a new object of that type
Read in and store the attributes into the new object
Register the new object with the world

For each instantiable class that you make, you must add in a new check to see if the class identifier that was read in matches the class identifier you have given your own class. If so, then you must dynamically allocate a new object of that type, read in and store the attributes into the new object, and finally register the new object with the world. That is exactly what the code in the while loop does. Also, note that this is the only place where we actually utilize a DraggableObject, so we needed to #include "draggableobject.h" at the top of worldio.cpp.

In a nutshell…

Congratulations! You have now read through the steps necessary to create a new instantiable class that exhibits interesting behaviours! As a recap, in order to create a new class with interesting behaviour, we have:

1. Figured out in explicit detail what we want our new object to do. This helps you decide which classes you need to derive from, and what methods should be overriden.
2. Implemented the object in its own specification/implementation file.
3. Wrote print/read methods that conform to the object definition format. In addition, the read method must be able to handle the special case where the class-identifier string is not there.
4. Modified the loadWorld function in worldio.cpp so that when a world file is read in draggable objects are instantiated.

Now you should follow these general steps in creating your own objects within this framework.

Commonly used objects and their methods

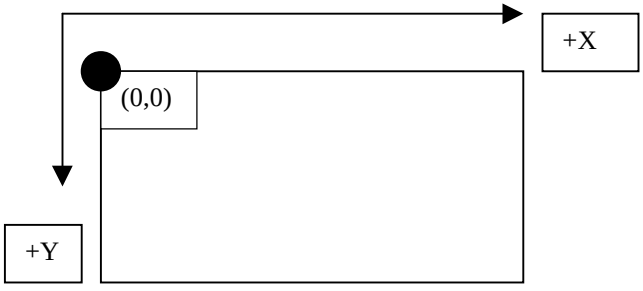
For all of the following descriptions, anything that is *italicized* should be considered to be a variable whose type is of the class that's currently being described.

Drawing Primitives

The following are the drawing primitives, along with some often-used method names and descriptions.

A word about the coordinate systems

The (x,y) coordinates used to specify points in the package has its origins in the upper left corner of the screen, with positive x going to the right, and positive y going down to the bottom of the screen.



Class DrawColor

This class is designed to be an abstraction for colors. A color is defined as a mixing of three colors: red, green, and blue. Each of these components are integers ranging from 0 to 255. The higher the value, the more intense that color stands out. For instance, red=255, green=0, blue=255 produces a magenta color. A color can also be defined in terms of a specific name, such as DrawColor::Red or DrawColor::Green. These two values are of type DrawColor::Name (see drawprimitives.h for a listing of all possible values of this type). The class has several member methods that allow for retrieving/storing colors.

Method Name	**Usage example**	**Purpose**
getByName	DrawColor::Name c=*color*.getByName();	Stores *color*'s value in c.
set(DrawColor::Name)	*color*.set(c);	Sets *color* to c.
getByRGBComponents(int, int, int)	*color*.getByRGB(r, g, b);	Stores the red, green, blue components of *color* to r, g, and b respectively
set(int, int, int)	*color*.set(r, g, b);	Sets *color*'s red, green and blue components to r, g, and b respectively

Class DrawPrimitive

All drawing primitives (except DrawColor) are derived from the DrawPrimitive class. All derived classes have the following methods associated with them:

Method Name	**Usage example**	**Purpose**
getColor	DrawColor c = *prim*.getColor()	Store *prim*'s color into c.
setColor(DrawColor)	*prim*.setColor(c)	Set *prim*'s color to c.
pointInPrimitive(DrawPoint)	bool isHit = *prim*.pointInPrimitive(pt)	If pt is within the bounds of *prim*, return true, else return false.
print(ostream&)	*prim*.print(out);	Print out *prim*
read(istream&)	*prim*.read(in);	Read in *prim*

Class DrawPoint

This class is designed to be an abstraction for points. A point is defined by its (x,y) coordinates and also has an optional radius. The class has several members that allow for point manipulation:

Method Name	**Usage example**	**Purpose**
getX	int x = *pt*.getX();	Store *pt*'s x coordinate in x.
getY	int y = *pt*.getY();	Store *pt*'s y coordinate in y.
getRadius	int rad = *pt*.getRadius();	Store *pt*'s radius in rad.
setX(int)	*pt*.setX(newX);	Set *pt*'s x coordinate to newX.
setY(int)	*pt*.setY(newY);	Set *pt*'s y coordinate to newY
setRadius(int)	*pt*.setRadius(newRadius);	Set *pt*'s radius to newRadius

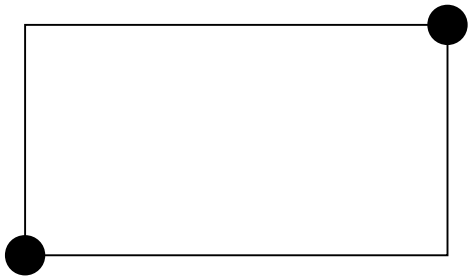
Class DrawLine

This class is designed to be an abstraction for lines. A line consists of two points: a beginning point, and an ending point, both of which are DrawPoints. A line may also have a width associated with it.

Method Name	**Usage example**	**Purpose**
getBegin	DrawPoint pt = *l*.getBegin();	Store *l*'s begin point in pt
getEnd	DrawPoint pt = *l*.getEnd();	Store *l*'s end point in pt
getWidth	int w = *l*.getWidth();	Store *l*'s width in w
setBegin(DrawPoint)	*l*.setBegin(pt);	Set *l*'s begin point to pt
setEnd(DrawPoint)	*l*.setEnd(pt);	Set *l*'s end point to pt
setWidth(int)	*l*.setWidth(newWidth);	Set *l*'s width to newWidth

Class DrawRectangle

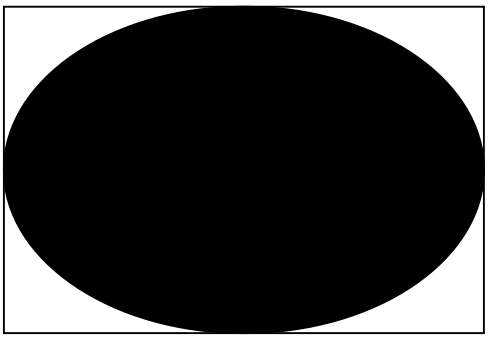
This class is designed to be an abstraction for rectangles. A rectangle consists of two points, representing opposite corners of the rectangle. A rectangle also may either be filled inside or not.



Method Name	Usage example	Purpose
getCorners(DrawPoint&, DrawPoint&)	*rect*.getCorners(pt1, pt2);	Store *rect*'s corners in pt1 and pt2.
setCorners(DrawPoint, DrawPoint)	*rect*.setCorners(pt1, pt2);	Set *rect*'s corners to pt1 and pt2.
getFill	bool isFilled = *rect*.getFill();	If *rect* is filled then return true, else return false.
setFill(bool)	*rect*.setFill(f);	Set *rect* to fill if f is true, else set *rect* to not fill.

Class DrawEllipse

This class is designed to be an abstraction for ellipses. An ellipse can be defined in terms of the rectangle that's the bounding box for the ellipse, like in the picture below. The ellipse can also be filled or not filled.



Method Name	Usage example	Purpose
getBounds(DrawRectangle&)	*el*.getBounds(rect);	Store bounding area of *el* in rect.
getFill	bool isFilled=*el*.getFill();	If *el* is filled then return true, else return false.
setBounds(DrawRectangle)	*el*.setBounds(rect);	Set *el*'s bounding area to rect.
setFill(bool)	*el*.setFill(f);	Set *el* to fill if f is true, else set *el* to not fill.

World Object

The world object receives and processes incoming events from the runtime system. The world keeps a list of updateable objects, and calls each updateable object's **update** method approximately once every tenth of a second. The world also keeps a list of drawable objects, and whenever the world needs to redraw the screen, each drawable object's **draw** method is called. The world also keeps a list of interactable objects. If user input is detected by the world, then each interactable object's method corresponding with that user input gets called (see the interactable object class for a list of supported user inputs).

The world also is in charge of loading/saving worlds from disk. In order to do this, two global functions, **loadWorld** and **saveWorld**, are called. The draggable object example talks in depth about the loadWorld function. The saveWorld function runs through all of the objects that registered with the world, and outputs their contents to the ostream that's passed in as a parameter.

The following is a list of methods that might be useful to programmers building instantiatable classes:

Method Name	**Usage example**	**Purpose**
getMousePoint	DrawPoint pt = *w*.getMousePoint();	Store *w*'s mouse position into pt.
getSize(int, int)	*w*.getSize(sx,sy);	Store *w*'s width and height into sx and sy, respectively
addObject(Object*)	*w*.addObject(objPtr);	Register objPtr with *w*, so that *w* can invoke the object's methods when needed.
removeObject(Object*)	*w*.removeObject(objPtr);	Unregisters the objPtr from *w*, so that *w* can no longer invoke the object's methods.
getAllObjects	ObjectList* li = *w*.getAllObjects();	Store a pointer to *w*'s object list in li.
update	*w*.update();	Inform *w* that an object has changed and needs to be redrawn

Note that the getAllObjects method, along with checking for specific types with isA (see object.h) is useful in trying to find and interact with objects of a specific type. This can be accomplished by iterating across the list looking for objects in which the object is a subclass of the specific type.

Class UpdateableObject

Updateable objects have one method of interest, called **update**, which takes in a pointer to a world object as a parameter. If the updateable object is registered with a world via the **addObject(Object*)** method, then the world will call the updateable object's update method approximately once every tenth of a second.

Class DrawableObject

A drawable object has associated with it an origin and a list of drawing primitives whose coordinates are relative to the origin. No matter where the origin is in the world, the drawing primitives will treat the origin point as (0,0) and its shapes will be drawn relative to that point. Suppose, for example, a drawable object has origin at (100,150) and the drawing list contains a point that will be drawn at (3,5). The point will actually be drawn (relative to the world) at (103,155).

A drawable object has one method that is called by the world the object registered with, the **draw** method. This method will draw all of the items in the draw list, taking into consideration the origin. This method generally does not need to be overriden by derived classes, though it is possible to do.

Drawable objects also have methods to set/get the origin, and to determine if a point is within the boundaries of the object.

Method Name	**Usage example**	**Purpose**
getOrigin(DrawPoint&)	*obj*.getOrigin(pt);	Store *obj*'s origin in pt
setOrigin(DrawPoint)	*obj*.setOrigin(pt);	Set *obj*'s origin to pt
intersects(DrawPoint)	bool collision=*obj*.intersects(pt);	Returns true if some primitive in *obj*'s draw list intersects with pt.

Class InteractableObject

InteractableObjects are objects that can respond to user input events, such as the clicking of the mouse button, and the movement of the mouse. If the programmer wishes to create an interactable object that handles a specific user event, like pressing the right button, then she will need to derive a new class from

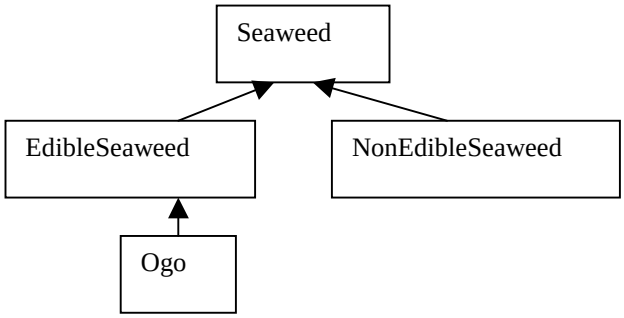
InteractableObject that overrides the appropriate method (in his case, the OnRButtonDown method). An interactable object that has registered with a world will have its method called if the world detects a user action. However, just because the world calls this method doesn't mean it's the object that the user is trying to manipulate. It is up to the individual method to determine if the user input is intended for that particular method (usually by determining if the mouse is clicking on the graphical representation of the object).

Each of these methods return a **boolean** value, which is **true** if the object decided to do something based on the user input (this is known as capturing the event), or **false** if the object decided not to do anything about it.

Method Name	**When called**
OnLButtonDown	User pressing down left button
OnLButtonUp	User releasing left button
OnRButtonDown	User pressing down right button
OnRButtonUp	User releasing right button
OnMouseMove	User moving the mouse

Using the isA function

In order to facilitate interaction with other objects, we provide a function called **isA**, defined in object.h, which takes in a pointer to an object, and a class name. If the object's type (Not the type of the pointer) is a subclass of the class name, then the function returns true. So, for example, let's suppose that we had the following hierarchy chart:



And the following definitions:
Seaweed sw;
EdibleSeaweed ed;
NonEdibleSeaweed non;
Ogo ogo;

Then the following chart lists some results that the isA function would give:

Call	**Result**
isA(&sw, Seaweed)	true
isA(&ogo, Seaweed)	true
isA(&ed, Ogo)	false
isA(&ogo, NonEdibleSeaweed)	false
isA(&non, EdibleSeaweed)	false
isA(&sw, Ogo)	false

The List class

This section describes one useful data structure that is used throughout the framework, the List class. Here are the methods, and its usage. The following chart should be used to determine what the type of item or pos should be, from the method name/usage chart:

Type of List	Type of item	Type of pos
ObjectList	Object*	ObjectList::Position
UpdateableObjectList	UpdateableObject*	UpdateableObjectList::Position
DrawableObjectList	DrawableObject*	DrawableObjectList::Position
InteractableObjectList	InteractableObject*	InteractableObjectList::Position
DrawPrimitiveList	DrawPrimitive*	DrawPrimitiveList::Position

Method Name	Usage/Purpose
isEmpty()	Checks to see if the list is empty
isFull()	Checks to see if the list is full
sizeOf()	Checks to see how many elements are in the list
data(pos)	Returns data at pos.
reset(pos)	Resets pos back to the head of the list
front(pos)	Places pos to the head of the list
back (pos)	Places pos at the end of the list
advance(pos)	Places pos at the next node in the list
previous(pos)	Places pos at the previous node in the list
endOfList(pos)	Checks to see if pos is off either end of the list
find(item)	Returns a position to the item in the list, or at endOfList if not found.
addToFront(item)	Inserts an item to the front of the list
addToEnd(item)	Inserts an item at the end of the list
insertBefore(pos, item)	Inserts an item just before the node pointed to by pos. pos will be pointing at the new node.
insertAfter (pos, item)	Inserts an item just after the node pointed to by pos. pos will be pointing at the new node.
deleteItem(pos)	Removes the node that pos is pointing to. pos will be pointing to the next item in the list.
clearAll()	Clears the entire list.
print(out)	Prints out the list.