2D Game Library User Guide

ARE 1.1

Eric Stone
Rayce Shurtz
Axle Fox

TABLE OF CONTENTS

Introduction

ARE is a 2D game library written in C++. It can be used to build cross-platform games for Mac, Linux, or Windows computers.

The library provides classes for all of the functionality you would expect to have in 2D arcade-style games, such as sound playback, text, image, and shape rendering, HUD elements, keyboard and mouse handling, physics, and collision detection. Classic games like Pong, Space Invaders, or Pinball are possible to create using only the features included in the library. With enough programming knowledge more complex games, like Mario or Metroid, are possible.

The collision system currently only supports 2D collision detection and physics, but 3D collision detection or other customized collision algorithms can easily be integrated. Graphics capabilities can also be extended with knowledge of OpenGL.

Installation Instructions

ARE Game Library Source - https://gitorious.org/gamelibrary_es

Required libraries:

Boost (v1.48) - http://www.boost.org/

Qt (v4.8) - http://qt.nokia.com/products/

OpenAL (v1.1) - http://connect.creativelabs.com/openal/default.aspx

Alure (v1.12) - http://kcat.strangesoft.net/alure.html

Compiling the library (Windows):

-Go to http://qt.nokia.com/downloads, then find "Qt libraries 4.8.0 for Windows (VS 2010, 275 MB)" and install it.

-Go to http://www.boostpro.com/download/, and click BoostPro 1.48.0 Installer.

-Install OpenAL from http://connect.creativelabs.com/openal/default.aspx. (look for OpenAL11CoreSDK)

-Install Alure from http://kcat.strangesoft.net/alure.html.

-Add the following entries to the PATH environment variable:

C:\Program Files\Microsoft Visual Studio 10.0\VC\bin;

C:\Qt\4.8.0\bin;

C:\Program Files\boost\boost_1_48_0\lib (or wherever you installed boost);

Path to OpenAL library

Path to Alure library

-Open the GBL_LIB.pro file and change the appropriate directories so that they match your computer's setup (i.e. if you use windows, modify the win32 section).

-Open the command prompt and navigate to the folder where the GBL_LIB.pro file is.

-Run "qmake -spec win32-msvc2010 -tp vc GBL_LIB.pro" in the command prompt.

-Open the GBL_LIB.vcxproj file that was created, and Build the project.

Compiling the sample games:

-In the .pro file for the game to compile, change the directories in the win32 section (assuming you're using a windows OS) to match paths to headers and libraries.

-Open the command prompt and navigate to the folder where the .pro file is.

-Run "qmake -spec win32-msvc2010 -tp vc project_filename.pro".

Getting Started

This section describes the fundamental parts of this library needed to get something up and running.

The first thing that must be done for any application is to create an instance of gameObj. There should only be one instance of this class per application.

```
gameObj MyGame(argc,argv);
```

You can also enable anti-aliasing by turning it on with the optional last parameter.

```
gameObj MyGame(argc,argv,GBL_MULTISAMPLE_ON);
```

The next thing is to create an instance of type gameComp. gameComp defines some of the basic window and game properties.

```
gameComp MyComp;
```

```
MyComp.setRefrate(30); // sets game loop to execute once every 30 milliseconds
MyComp.setWindim(400,400); // sets window dimensions to be 400 x 400 pixels
MyComp.setWinpos(0,0); /* sets the windows top left position to be at the very top left
        of the screen*/
MyComp.setBGColor(1,1,1); // sets the default background color to be white
MyComp.setScale(40,40); /* (See diagram on next page) This sets how much the screen
is scaled by. A screen size of  400x400 with a scale of 40x40 will have a ratio of 10:1.
This means that items will be 10x their normal size if drawn using their normal
dimensions. The position is scaled as well. An item positioned at (5,6) would have screen
coordinates of (50,60) when drawn. You can set the scale to equal the windim value for
a 1:1 ratio.*/
```

At this point, you would prepare your game. Create 2D objects, set up mouse/keyboard event handling, set up collision system, etc. These are described in other sections.

The main loop does not start until the exec function is called. This means mouse/key events, collisions, and even the main window will not actually be created until this function is called. The loop will continue running until the main game window is closed, and this function will not return until that happens. At this point the library internals take control of program execution. Total control is not lost though; there are ways to get control at certain times, the main way of doing this are through keyboard/mouse events, and through the miscAction system. These are described in other sections.

To start the main loop, pass the gameComp instance to MyGame.exec() function:

```
MyGame.exec( MyComp );
```

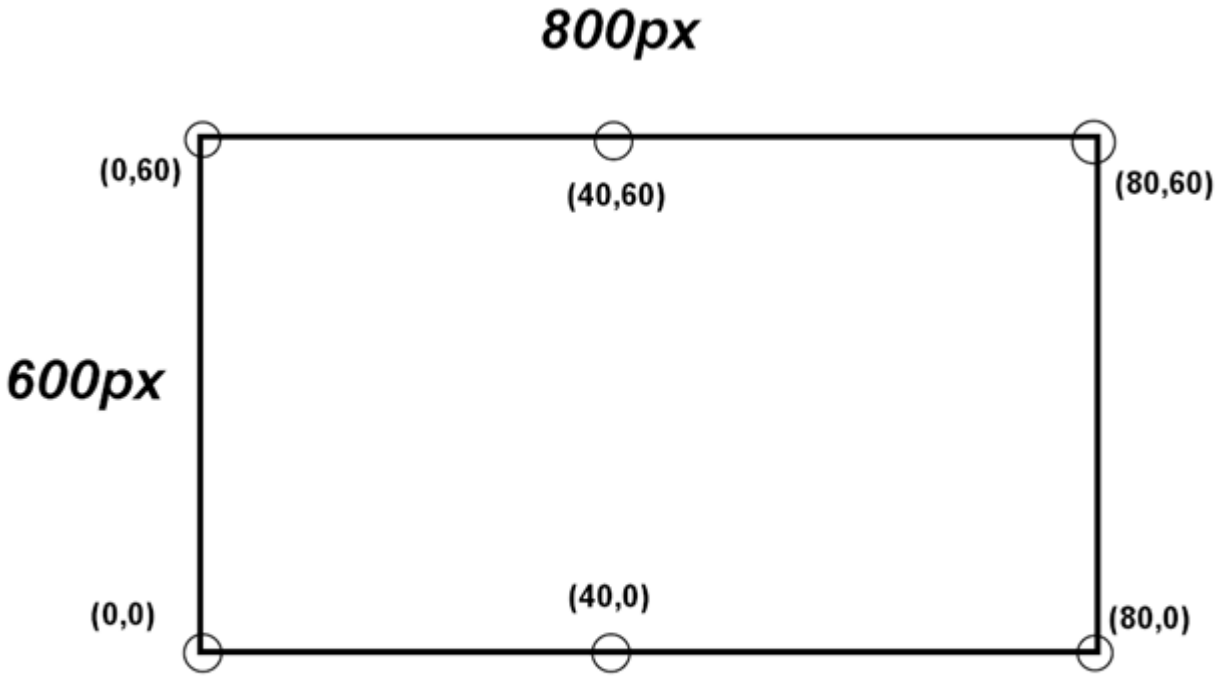
At this point the game window opens, the main loop starts executing, and mouse/key events will be processed.

If you want the game to exit, you should call:

```
gameInfo::quit();
```

This needs to be called in order to do some cleanup before the app closes.

Window Scale Diagram



Window Dimensions: 800x600
Window Scale: 80x60

Managing Game States

The gameState class is used to save objects specific to the state. It keeps track of miscAction, text, images, drawing, timers, the mouse handler, and the keyboard handler. You can easily switch states at runtime.

To create a new state:

```
gameState MyState1;
```

There are four main objects that can be added or removed from a state. These are miscAction, gblText, gblImage, and drawBasel objects. The mouse and keyboard handlers are also set to a specific state.

```
gblImage Background("/opt/image/bg.png", 0,0, 80,60, 0);
MyState1.addToScene(&MyImage);
MyState1.remove(&MyImage);

boost::shared_ptr<MyMouseClass> MouseHandler(new MyMouseHandlerClass);
MyState1.setMouseHandler(MouseHandler);
```

When an object that is in a state (or multiple states) is destroyed it will automatically remove itself from any states it was in, so you don't have to manually remove it before you delete it.

```
gblImage *Background = new gblImage("/opt/image/bg.png", 0,0, 80,60, 0);
MyState1.addMisc(&MyImage);

delete Background;
// MyState1 no longer has a pointer to Background
```

To set the current state:

```
gameInfo::setCurrentState(&MyState1);
```

Note: The current state will not actually be changed until the end of the current loop.

Types and Lists

This section introduces some important classes along with a short description and important properties. The reference document contains a full list of constructors and public members.

I. Basic 2D types

point2f - 2D points.

.

```
point2f MyPoint(20.5,11.5); // x=20.5, y=11.5
```

float x – X value of the point
float y – Y value of the point

circle - 2D circles.

```
circle MyCirc(2,20,20,.2,-.2); // radius=2, tloc=(20,20), shift=(.2,-.2)
```

float radius - Radius of the circle
point2f tloc - The location of the center of the circle as a point2f
point2f shift - The velocity of the circle as a point2f

dwall2f - 2D lines.

```
dwall2f MyLine dwall2f(0,2,40,2,.2,-.2); // p1=(0,2), p2=(40,20), shift=(.2,-.2)
```

point2f p1 - The first point of the line
point2f p2 - The second point of the line
point2f shift - The velocity of the line as a point2f

polyPoint - 2D polygons.

```
polyPoint MyPoly polyPoint;
```

vector<point2f> ps - All points of this line.
point2f shift - The velocity of the polygon as a point2f

Use polyPoint::addPoint(point2f) function to add points to the polygon:

```
MyPoly.addPoint(2,5);
```

The shapeUtil namespace has many functions for creating, manipulating or getting information about shapes. Includes function for translation, rotation, creating common polygon shapes, get/set min/max x or y value, and others.

II. Lists

Having a list of elements is often useful when multiple shapes are used in the same manner. Some parts of this library will only work with a list of shapes, such as the collision system and some parts of the drawing system. There are two types of list classes, es_list and multiCont. They work in nearly the same way but multiCont has extended functionality which allows more complex drawing.

es_list<T>

This is a template class, so can be used to store a list of any type. Basic usage is similar to the standard template library's std::list, the main difference is that objects are added to the list by passing a pointer to the push_back function instead of by reference.

To create an empty list, in this case a list of circles:

```
es_list<circle> MyList;
```

To add elements to the list:

```
MyList.push_back( new circle(2, 1,1,.2,.2) );
```

Access to elements requires the use of an iterator, to declare one:

```
es_list<circle>::iterator ii;
```

To set the iterator to point to the first element in the list:

```
ii = MyList.begin();
```

The -> operator can be used to access the members of the object.

```
ii->shift.x = .5;
```

The * operator is used to dereference the object. It returns a reference to the element.

```
(*ii).shift.x = .5;
```

Iterators can be incremented using the ++ operator:

```
++ii;  // this now points to the next element in the list, if it exists
```

It is IMPORTANT to check that an iterator has not reached the end of the list:

```
if( ii == MyList.end() )  /*If this statement is true, the iterator has reached the end of the
list, and trying to access ii here will result in undefined behavior.*/
```

You can also get a reference to the first element by using the front() function:

```
MyList.front().shift.x = .5;
```

To erase an element from the list, you pass an iterator to the element to be deleted into the erase() function:

```
MyList.erase( ii );
```

Keep in mind, comparing two different iterators will be true if they point to the same data, they do not compare the data values. If you want to compare the values themselves then use the dereference (*) operator.

```
if( ii1 == ii2 ) // this statement is only true when iterators point to the exact same data
    ii1->shift.x = .8;  // ii2->shift.x now also equals .8
if( *ii1 == *ii2 )
    cout<< "The value in ii1 is equal to the value in ii2"<<endl;
```

multiCont<T>

The multiCont container inherits everything from es_list, so you can use it the same way as a list, and any function that accept an es_list pointer will also accept a multiCont pointer. In addition though, it contains a "drawCont" object. The drawCont defines how elements in the list will be drawn. All elements in the list will be drawn the same, as defined in the drawCont. The drawCont class will be described more in the drawing section of the documentation.

The main purpose of this class is to allow the collision system to handle objects as a simple shape (circle/polyPoint/or dwall2f), but the objects will be appear more complex and could include multiple different types of shapes and colors.

The constructor of this requires a drawCont pointer.

```
drawCont *MyDrawing = new drawCont;
//.....add stuff to MyDrawing (described later)
multiCont<circle> MyCont( MyDrawing );
```

You can change the drawing by calling the setDrawCont() function, and passing a different drawCont instance:

```
drawCont *MyOtherDrawing = new drawCont;
//.....add stuff to MyOtherDrawing (described later)
MyCont.setDrawCont( MyOtherDrawing );
```

Don't forget to add this to the drawing scene:

```
MyState1.addToScene( MyDrawing );
```

The miscAction System

This is way to get control of execution during runtime. Each miscAction object added to the game will have their action() function called within the main loop on every loop. Many components of this library, such as the collision system, are derived from the miscAction class. You can create your own class that publicly inherits from the miscAction class if you want your code to execute every loop.

The miscAction system is fairly simple, but an important tool to understand. It is used to for miscellaneous functions that need to be called on every loop. For instance, a game might want to check the score and end the game if a player finished a level, or won the game. Using this system, you can do this by checking the score on each loop, and perform necessary actions if needed.

For miscellaneous, game specific code that is unrelated to mouse/key events, this is a good place to put it. Once you get it set up, it will be called once each time the main loop is executed. You may create as many miscAction objects as needed.

The first step to use this system is to create a new class which publicly inherits from the miscAction class. This class Must have a member function "void action()".

```
class MyClass : public miscAction
{
        int MyInt;
public:
        MyClass() : MyInt(0){}
        void action()
        {
                std::cout<<"MyInt = "<<MyInt<<std::endl;
                ++MyInt
        }
};
```

The second, and last, step is to add an instance of your class to any gameState's that use this object:

```
MyClass MyInstance;
MyState1.addMisc( &MyInstance );
```

MyInstance->action() will be called once every loop.

Mouse Events

```
struct mhBase{
	virtual void pressed(point2f aloc);//called once when mouse button is pressed
	virtual void released(point2f aloc); //called once when mouse button is released
	virtual void moved(point2f aloc);//called when mouse is moving AND mouse is unpressed
	virtual void dragged(point2f aloc); //called when mouse moves AND a mouse button is clicked
	virtual void idle(point2f aloc);//called when mouse is stationary AND button unpressed
};
```

This is the class that is used to create your own mouse handler. You inherit from this class, and implement the virtual functions in the derived class with the functionality you want. Here's an example of how it might look:

```
class MyMouseHandler : public mhBase{
public:
	MyMouseHandler(){}
	void pressed(point2f aloc){
			//shoot a laser
		}
	void released(point2f aloc){
			//stop shooting/do nothing
		}
	void moved(point2f aloc){
			//player ship moves towards cursor
		}
	void dragged(point2f aloc){
			//drag player ship to fling it when button is released
		}
	void idle(point2f aloc){
			//do nothing/gradually restore energy/etc
		}
};
```

The uses are as varied as the programs you can create with the library. Your mouse handler can move images, display text, play sounds, call functions, and so on. Values can be passed into the constructor for extra control, and you can create as many other functions as you want in your mouse handler class. The example provided is how an arcade-style spaceship shooter like Galaga, Asteroids, or Gradius might behave. Each function receives a point2f which stores the current location of the mouse. You can also get the current position of the mouse at any time by calling the static function escursor::mouseLoc() function, which returns a point2f.

To set your application's mouse handler, you pass a boost::shared_ptr object containing an object of your mouse handler class to any states that use this mouse handler.

```
boost::shared_ptr<MyMouseHandler> MousePtr(new MyMouseHandler);
MyState1.setMouseHandler(MousePtr);
```

The MyMouseHandler instance will receive mouse events to its virtual functions.

Keyboard Events

Keyboard event handling works in a similar way to mouse event handling. You inherit from the abstract class keyBase, and re-implement the keypress and keyrelease functions. Here is an example (see the example at the end of this document for more information):

```
class myKey : public keyBase
{   public:
        void keypress(esKeyEvent keyEvent)
        {       switch( keyEvent.getKey() )
                {
                case 'w' :
                        YellowBalls->front().shift.y +=2;
                        break;
                case 's' :
                        YellowBalls->front().shift.y -=2;
                        break;
                case 'a' :
                        YellowBalls->front().shift.x -=2;
                        break;
                case 'd' :
                        YellowBalls->front().shift.x +=2;
                        break;
                }
        }
        void keyrelease(esKeyEvent keyEvent)
        {
        }
        myKey(es_list<circle> *yballs){YellowBalls=yballs;}
    private:
        es_list<circle> *YellowBalls;
};
```

Add the key handler to any gameState's that use it:

```
boost::shared_ptr<myKey> TestKey(new myKey(&YellowBalls));
MyState1.setKeyHandler(TestKey);
```

Text

```
gblText MyText(0,0,2,"Your Text Here");
gblColor aColor(20,45,10);
MyText.setColor(aColor);
MyText.setFont("Times",80);

MyState1.addToScene(&MyText);
```

To display text on screen, you must first create a text object. This object will store all of the information about your text, such as the font, color, position, etc. In this example, the first item is the creation of a text object called MyText. The text is created with some attributes, with the parameters corresponding to x, y, layer (or depth), and the text you want to display. You can also leave it blank and pass no parameters, then use the provided functions to set the different attributes of your text object later. Even if you do pass in parameters for you text when the object is created, you can still use the provided functions to change the text object at any time.

The second item is a color object, which is used in the next line to set the color of the text. The RGBA values of your color are float values with a range of 0-1. After that is a function that allows you to set the font and font size of your text. The font, size, and color all have default values, so it's not required that you set them.

There are also a few other functions that can be used to set (or change) the attributes of your text object:

```
MyText->setPos(10,10);//set the position of the text
MyText->setLayer(1);//set the layer (or depth) of the text
MyText->setText("Your Text Here");//specify the text to be displayed by this text object
```

The setPos function allows you to specify the position you want your text to be displayed at. This function, as well as the others, overrides the pre-existing values you may have set for it. If you create a text object at position 5, 8, then use setPos(10,10), your text will display only at the newer position (and so on for the other text attributes). When setting the position with either the setPos function or by passing in parameters when creating a new text object, a single point2f can take the place of both the x and y parameters.

The setLayer function is used to specify the depth of your text, which is used for displaying it on top of other items. 0 is the default.

Last of all is the setText function, which allows you to specify the text you want to display. If your text object already has some text you've previously specified, it will be overwritten. To display text on multiple lines, multiple text objects are required.

Images

```
gblImage *image = new gblImage("X:/filepath/imagename.filetype",0,0,7.3,7.3,2);
MyState1.addToScene(image);
```

This is the simplest syntax for displaying an image. The first item is the creation of a new image object, with parameters that correspond to the image's filepath, x, y, width, height, and layer (or depth). The width and height are relative to the ratio between the working space and the screen size.

Example: If you have a screen size of 800x600, but a working space of 80x60 (the value set with gameComp.setScale), the ratio is 10:1. This means that the image width and height values would need to be 10 times smaller than the actual image size for it to display the image at its normal size. If the image was 10x10, you would need to specify a width and height of 1x1 to achieve this. If you specified 10x10 while the ratio is 10:1, the image would be stretched to 100x100 pixels. A 1:1 ratio of screen size to workspace would allow you to use the literal image dimensions as the width and height of the image object.

The second item is the function that adds the image to a list of images to display on screen. It works in a very similar manner to how text is handled. The other functions you can use to set image properties have a similar syntax to that of text as well:

```
image->setPos(10,10);//set the position of the image
image->setDim(10,10);//set the dimensions of the image
image->setLayer(1);//set the layer (or depth) of the image
```

The first item, setPos, works in exactly the same way it does for text. You can pass in either an x and y value, or you can use a point2f.

The setDim function can accept the same parameters as setPos: x and y, or a point2f value. It's used to specify the image's size (or resize the image if it already has different dimensions).

The last item, setLayer, is identical in syntax and function to the one used for text.

When creating the image object, you can either specify some attributes for the image (as shown in the example of the simple two line image creation method), or leave the parameters blank and specify the attributes with the provided functions. Anything specified with the provided functions will overwrite whatever may have been specified previously. Other acceptable parameters when creating an image object include: passing in only a filepath, an image object by reference, and a pointer to an image object. There is also a final set of acceptable parameters which is a minor variation to the one used in the example: two point2fs can be used to replace the x, y, width, and height values. The first point2f corresponds to the x and y position, and the second corresponds to the width and height of the image. The other parameters all remain the same as in the example. You must either use both point2fs or all four individual parameters they represent (x, y, width, height) along with the rest of the parameters.

Sound

There are currently three different classes available for playing audio, gblSound, gblAudio, and gblMusic. gblSound and gblAudio are intended for sound effects, gblMusic is intended for playing background music.

Note: gblSound is not recommended, but still included. It is slower than gblAudio and can only play one sound at a time.

gblSound and gblAudio

These classes should be used for simple sound effects. They only officially support the .wav file format, but may be able to play other formats. Both classes are used exactly the same way.

To create a gblSound or gblAudio object, pass the path and name of the file in the constructor:

```
gblAudio MyAudio("/opt/mySound.wav");
gblSound MySound( "/opt/mySound.wav" );
```

On Windows computers, you must use forward slashes in the path, or double backslash.

```
gblSound MySound("C:\\gameFolder\\mySound.wav");
```

To change the volume:

```
MyAudio.setVolume(.5);
```

To play the file:

```
MyAudio.play();
MySound.play();
```

gblMusic

This class was created as a simple way to play background music. Only one gblMusic object should be created at a time. This class allows starting/stopping/pausing a song, adding songs to the play queue, and skipping songs. Only one song may be playing at any one time.

To create a gblMusic object:

```
gblMusic MyMusic("C:\\gameFolder\\myMusic.mp3");
```

To play the file:

```
MyMusic.play();
```

To add another music file to the queue, this file will start immediately after the previous song finishes playing, or if nothing is currently playing it will begin immediately:

```
MyMusic.enqueue("C:\\gameFolder\\addAnother.mp3");
```

To skip current song and start playing next song in the queue:

```
MyMusic.next();
```

To change the volume:

```
MyMusic.setVolume(.5);
```

Note: the gblMusic object must be created after the application's gameObj instance is created.

Score Meters

There are several classes provided that help with score keeping and displaying the score.

The class used to keep track of a score is the meter class. When you create this class, you specify a minimum, maximum, and current value.

```
meter MyMeter( 0, 10, 8); // min=0,max=10, current=8
```

To change the current value, use the meter::add(int) or meter::setCur(int) function.

```
MyMeter.add(-3); // reduce current value by 3
MyMeter.setCur( 10 ); // set the current value to 10
```

Both the add() and setCur() functions will make sure the current value never goes beyond the min or max values.

To get the current value, and change the min/max:

```
int cur = MyMeter.Cur();
MyMeter.setMin( -5 );
MyMeter.setMax( 15);
```

Displaying Meters

There are several convenient HUD classes to easily display the meter:

- Horizontal Bar - Meter is shown as a horizontal bar.
- Vertical Bar - Meter is shown as a vertical bar.
- Circle Bar - Meter is shown as a circular bar.
- Number Bar - Shows the meter's current value displayed as a number.
- Shape Bar - A shape is shown for each positive integer.

To use the display bars, create the HUD type you want, passing a pointer to your meter in the constructor, as well as other parameters that control how the HUD is displayed. Examples:

```
hudHBar MyHorizontalBar(&MyMeter, 10, 40, 20, 3, gblColor::red, 0 );
/* bottom left corner at (10, 40), width 20 when meter is full, height 3, color red.
The last parameter 0 means min is on left, max is right.  1 would do the opposite.*/
hudVBar MyVerticalBar(&MyMeter, 10, 40, 20, 3, gblColor::red, 0);
// bottom left corner at (10,40), height 20 when full, red, min on left, max on right
hudCircBar MyCircularBar(&MyMeter,10,40,4,8,gblColor::red,0);
// bottom left corner at (10,40), radius of 4, thickness of 8, red, direction (left or right)
hudNumberBar MyNumberBar(&MyMeter,10,40,gblColor::red, "Times", 30);
// bottom left corner at (10,40), red, Times font, font size 30
drawCont MyDrawContainer;
// refer to drawCont in the Drawing System section for more info...
hudShapeBar MyShapeBar(&MyMeter, &MyDrawContainer, 10, 40, 5, 0);
// bottom left at (10, 40), width of each element is 5
```

```
// 0 means min elements on left, max on right

MyState1.addToScene( &MyVerticalBar );
```

Timers

gblTimer

This class is a basic timer (not high precision). It can be used to execute a function once a time limit is reached.

To create it:

```
gblTimer MyTimer(500);
```

The constructor argument is an amount of time specified in milliseconds. But there are also two convenient functions that will convert seconds or minutes to milliseconds:

```
gblTimer MyTimer(timSeconds(.5)); // will expire in half a second, same as 500 millisec
gblTimer MyTimer2(timMinutes(.5)); // will expire in 30 seconds
```

The timer has not started counting down yet. We have two more steps.
Connect a function object:

```
//assuming a class named MyClass with member function "void myFunc()" is defined
MyClass *MyInstance = new MyClass;
MyTimer.setFunction( boost::bind( &MyClass::myFunc, MyInstance ) );
```

The timer will now start counting if the start function is called. If you call the start() function now, the timer will expire once and be done (you can call start() again later if you want to though).

Now that a function is connected, we have to start the timer:

```
MyTimer.start();
```

A timer can be stopped at any time:

```
MyTimer.stop(); // the connected function will NOT be called
```

Note: The timer will only work correctly while the main loop is running, or after your gameObj's exec() function is called.

A timer will be reset when it expires or when stop() is called.

Networking

Client/server networking can be achieved using the netServer and netClient classes. This will work on a LAN.

There is a simple example client and server program in the sampleGames repository. Note that you would need to change the ip addresses in the code to match your actual ip addresses. Keys typed on the client side will be displayed on the server side. Keys typed on the server side will be displayed on the client side.

Starting the Server

```
void serverReceive(const char *Data, const std::string& ip)
{
    std::cout<<"Server Rec: "<<" : "<<Data<<std::endl;
}

int main(int argc, char *argv[])
{

boost::function<void (const char*, const std::string&)> SFunc
                        = boost::bind(serverReceive,_1,_2);

netServer MyServer(SFunc);

if(!MyServer.setConnectionInfo("192.168.1.30",35022,46011))
      cout<<"Error :"<< MyServer.getErrorMessage()<<endl;
else if(!MyServer.start())
      cout<<"Error :"<< MyServer.getErrorMessage()<<endl;
```

The first line in the main function will bind the serverReceive function to a function object.

The second line constructs a netServer object and passes the function object to the constructor.

The function object will be called any time a client sends data using the member function netClient::sendtoServer. The Data will be a character array, and the string in the second parameter will contain the IP address of the sender.

The next part attempts to set the server connection info. The MyServer.setConnectionInfo(...) tells netServer that the server should be run on 192.168.1.30 and it should listen for incoming data on port 35022 and will write data to port 46011. That function will return 0 if a problem was encountered.

The MyServer.start() will attempt to bring the server online so it can start reading or writing data. It will return 0 if a problem encountered.

Starting a Client

A client is started in a similar way:

```
void clientReceive(const char *Data)
{
    std::cout<<"Client Rec: "<<Data<<std::endl;
}

int main(int argc, char *argv[])
{

boost::function<void (const char*)> CFunc
                        = boost::bind(clientReceive,_1);

MyClient.start();//

if(!MyClient.setConnectionInfo("192.168.1.25","192.168.1.30",46011,
35022))
      cout<<"Error :"<< MyClient.getErrorMessage()<<endl;
else if(!MyClient.start())
      cout<<"Error :"<< MyClient.getErrorMessage()<<endl;
```

You'll notice that, unlike netServer example, the clientReceive function only takes one argument. Also, the netClient::setConnectionInfo function takes has an extra parameter. The first one is the client's IP, the second is the server IP. The third paramenter is the port the client will listen for data on. The fourth one is the port it will write to. Notice that the client read port is the same as the server write port, and that the client write port is the same as the server read port.

Note that if MyClient.start() returns 1, it only means the client can successfully read/write data to a socket, but it does not mean that it has connected to the server. It will attempt to communicate and connect to the server at this point. The MyClient.getTeleTimeOff() will return the number of seconds since it last received a communication from the server, or if it has not received any data from the server at all then it will be the number of seconds since MyClient.start() was called.

Sending Data

If all went well with setting up the connections, data can now be sent between clients. Currently, clients can send data 3 ways:

```
MyClient.sendVerify("Make sure this gets there!");
```

This will send data to all the clients. It will ensure the data gets to all clients. It is sent over a UDP socket but netClient and netServer will keep track of the packet and make sure it gets where it needs to go. Multiple packets sent over this function may not arrive in the same order they were sent.

```
MyClient.sendNoverify("Send and hope this gets there!");
```

This function simply sends the data over a UDP socket to all clients connected to the server, and does not verify that it will get to its destination.

```
MyClient.sendtoServer("Send this to the server");
```

This will send the data to the server, it will not be sent to any clients. The data will be tracked so it should get to the server whenever you send it this way.

Drawing System

I. Drawing shapes that are in an es_list

There are convenient ways to draw lists of circle, dwall2f, and polyPoint. The process is the same for all three. They can be drawn as filled, or just the outline of them (dwall2f can only be drawn as a line).

The code below will draw the circles in WhiteCirc list as white filled circles, on layer 0. The circles in GreenCirc will only have their outlines drawn, will be light green and on layer 1.

```
es_list<circle> WhiteCirc;
es_list<circle> GreenCirc;
//... add circles...
MyState1.addToScene( drawFactory::drawCirc( &WhiteCirc, gblColor::white, 0 ) );
MyState1.addToScene( drawFactory::drawCircOutline( &GreenCirc, gblColor::lgreen, 1 ) );
```

If the list were of polyPoints instead of circles, the calls would be similar. Instead of 'drawCirc' and 'drawCircOutline' in the example of above, you would replace it with drawPoly or drawPolyOutline. For a list of dwall2f, you would call drawLine instead.

Note: drawPoly() may not draw polygons with concave sides correctly. This is due to a limitation with OpenGL.

II. Drawing simple shapes that are not in a list

If you want to simply draw a shape whose position and other attributes will not change throughout the game, you can call these functions:

```
MyGame.addToScene( drawFactory::RectangleFill( 0,0,10,5, gblColor::white, 0 ) );
   // ^ Create rectangle whose bottom left point is (0,0), and top right is (10,5), layer=0
MyGame.addToScene( drawFactory::CircleFill( 2, point2f(11,11), gblColor::black, 2 ) );
   // ^ Create circle with radius=2, center at (11,11), color black, and layer 2
MyGame.addToScene( drawFactory::Line ( 5,5,20,25, gblColor::lgreen, -1 ) );
   // ^Create line where first point is at (5,5), second at (20,25), color light green, layer -1

polyPoint MyPoly;
MyPoly.addPoint(0,0);
MyPoly.addPoint(10,0);
MyPoly.addPoint(5,7);
MyState1.addToScene( drawFactory::PolyFill( MyPoly, gblColor::white, 0 ) );
MyState1.addToScene( drawFactory::PolyLine( MyPoly, gblColor::lgreen, 0 ) );
```

Keep in mind that using the above method will not allow you to change the position of the shape or anything else. Best used for objects that don't need to change throughout the duration of the game.

Note: PolyFill may not draw polygons with concave sides correctly. This is due to a limitation with OpenGL.

III. drawCont

The main purpose of the drawCont class is to allow a group of multiple shapes/colors to be handled as a single unit, allowing more complex figures to be drawn that may be composed of many different circles, lines, polygons, and/or multiple colors.

You use the same functions that were used in the previous section to add shapes to the container. Instead of calling them in Mygame.addToScene(), you call the drawCont object's add() function.

```
drawCont *MyDrawContainer = new drawCont;
MyDrawContainer->add( drawFactory::RectangleFill( 0,0,10,5, gblColor::white, 0 ) );
MyDrawContainer->add( drawFactory::CircleFill( 2, point2f(5,2), gblColor::lgreen, 2 ) );
```

The x,y coordinates specified above are relative to the drawCont's 'Reference Point' variable. The drawCont's Reference Point can be changed during the game:

```
MyDrawContainer->setRefPoint( 20,20 );
```

Don't forget to add the drawCont to the scene!:

```
MyState1.addToScene( MyDrawContainer );
```

Using drawCont with multiCont<T>

A drawCont is very useful with a multiCont container. When used in this way, the drawCont's Reference Point is changed automatically, so you do not have to worry about setting it. Just remember that if the multiCont is storing circles, the Reference Point will be changed to the center of the circle for each circle. For polyPoint, it will be the first point in the polygon, and for a dwall2f it will be the first point of the line.

```
multiCont<polyPoint> *MyPolyLst = new multiCont<polyPoint>( MyDrawContainer );
MyPolyLst->push_back( shapeUtil::box2f( 30,30,40,35 ) );
    /*the shapeUtil::box2f() function is a convenience function. It creates a polyPoint
    with four points, in the shape of a rectangle*/
MyPolyLst->push_back( shapeUtil::box2f( 10,10,20,15) );
```

In the above example, MyPolyLst can be used with the collision system, where elements in it will be treated as simple polygons, but when they are drawn they will be drawn using the drawCont. This allows more complex objects to be drawn for objects, but the collision system will still handle it as a simple shape.

IV. Drawing with Raw OpenGL code

If you want to draw stuff using raw OpenGL code, the drawBasel class is provided. Inherit from drawBasel and implement the draw() function with your OpenGL code.

```
class MyDrawClass : public drawBasel
{
public:
    void draw()
    {
      glPushMatrix();
       glTranslate( 70.0, 35.0, 0);
          glBegin(GL_TRIANGLE_FAN);
             glColor3f( .2, .2, .8);
            glVertex3f( 5.0, 5.0, 0);
            glVertex3f( 5.0, 0.0, 0);
            glVertex3f( 7.0, 2.0, 0);
              glColor3f( .8, .2, .2);
            glVertex3f( 10.0, 5.0, 0);
            glVertex3f( 7.0, 7.0, 0);
            glVertex3f( 5.0,10.0,0);
            glVertex3f( 3.0, 7.0, 0);
          glEnd();
       glPopMatrix();
    }
};
```

You can add an instance to the scene, or add it to a drawCont container:

```
MyDrawClass MyInstance;
MyState1.addToScene( &MyInstance );

drawCont MyDrawContainer;
MyDrawContainer.add( &MyInstance );
MyState1.addToScene( &MyDrawContainer );
```

Collision System

To use the collision system, you must first create an object of colMachine:

```
colMachine MyCol;
```

Then add the object to the game by passing it to a gameState's addMisc() function:

```
MyState.addMisc(&MyCol);
```

The collision system works only with lists of shapes stored in an es_list. It supports circles, dwall2fs, and polyPoints, or any objects derived from these.

You also need to define what will happen when a collision occurs. You may want them to bounce off each other, only one of them to bounce, or you may want one or both elements to be deleted.

When adding to the collision system, you create a "gear". You can think of the gear as part of the collision machine that stores the two objects (it actually stores two lists of objects) where each element in the first will be tested for collision against the every element in the second. Each gear also defines the behavior each object will exhibit when a collision occurs.

Here is an example of creating a gear. In this example we will simply have two balls that will bounce off each other when they collide, and will bounce when it hits the edges of the screen. It is assumed you have read the document section, which describes the circle, dwall2f, and es_list classes. You may also want to view tutorial1, which describes creating shapes, getting them drawn, and adding them to the collision system.

This example creates two gears. The first specifies that elements in MyBalls list will bounce when they collide with elements in MyWalls, while the MyWalls list will do nothing. The Noth keyword means that the velocity of the object will not change. If it is moving, then it will continue moving in the same direction at the same speed. If it is stationary, it will stay that way. The second gear makes the circles in MyBalls list bounce off each other.

```
es_list<circle> Myballs;
es_list<dwall2f> MyWalls;
MyCol.addGear( gearFactory::makeGear( Bounce(&MyBalls), Noth(&MyWalls) ));
MyCol.addGear( gearFactory::makeGear( Bounce(&MyBalls) ));
```

There are four types of behaviors available. Bounce indicates the object should bounce when it collides, Noth indicates it's velocity will not change. Del indicates the element will be deleted. Stop indicates it will stop moving and it's velocity will be set to (0,0). Only certain combinations are supported, they are listed in the table below.

Available when two lists are involved: A '+' indicates support.

	Bounce	Noth	Del	Stop
Bounce	+	+	+	
Noth	+	+*	+	
Del	+	+	+	
Stop				+

* Only limited support available for Noth/Noth. Both objects Must be either circles or boxes (boxes are polyPoints created with shapeUtil::box2f()).

Where single gears are used, as in second example above, only Bounce, Del, and Stop are supported. Noth is not supported because the collision system is used to ensure that objects never overlap by changing their velocity or deleted them when collisions occur.

Important: Keep in mind that the collision system will change the location of objects that use it. Extreme caution should be taken when changing the location of objects manually yourself (by changing circle's tloc variable, dwall2f's p1 or p2 variables, or any of polyPoint's ps variables). This is because you may accidentally move an object somewhere that is already occupied by something else. You can always change the velocity of objects by setting it's 'shift' variable, but it is best to let the collision system change the location of the object.

What if a program needs more specialized behavior when a collision occurs? The game state may need to be changed in some way. The way to do this is to pass a boost::function object as a third parameter for the gearFactory::makeGear() function. The function must return void and take two parameters. The specific 2 elements that collide will be passed to this function when a collision occurs. The first parameter must be of the same type of elements in the first es_list added to the gear, the second must be the same type as the elements of the second es_list added to the gear. Both are passed by reference. If you are not familiar with creating a boost::function object, here is an example.

First you need a function. Here we make a new class with a member function. Notice the function returns void, first parameter takes a reference to a circle, and the second takes a reference to a dwall2f.

```
class MyClass
{
  public:
    void BallWallCollision( circle& acirc, dwall2f& awall)
    {
        std::cout<<"A Collision Occurred!!"<<std::endl;
    }
};
```

Now to create a boost::function object, and bind the function and object:

```
MyClass MyInstance;   // first we need an instance of the class
boost::function<void (circle&,dwall2f&)> MyFunctionObject
        = boost::bind( &MyClass::BallWallCollision, &MyInstance, _1, _2 );
```

```
MyCol.addGear( gearFactory::makeGear( Bounce(&MyBalls), Noth(&MyWalls),
MyFunctionObject ));
```

The call above is the same as earlier, except with the function object passed as the third parameter. When a collision occurs, it will basically call MyInstance.BallWallCollision(Circ1, Wall1), where Circ1 is the specific ball, and Wall1 is the specific wall.

The BallWallCollision function simply prints to the console whenever a collision between elements in MyBalls and MyWalls occur. But could easily do other things like increase a score, end the game, add new elements to the game, etc…

The same method can be used with the single gear objects. The function created would still have two parameters, both with the same type. The function object would be created the same way, and passed to the gearFactory::makeGear() function as the second argument.

Specialized Collision Detection for Custom Shapes (like 3D collision)

Section not completed yet...Basically you create a specialized template class for the colCheck class and make sure to set the necessary variables that the collision system needs.

Utility classes

friction

To cause your objects to slow down, use the friction class.
There are two ways to use it. First, with any shape, you specify a value as the second parameter. Second, if it is a shapePhys then a friction value is already stored within shapePhys object. Use a value between 0 and 1. More friction is applied the closer it is to 0. A 1 will apply no friction to the object.

With any shape, where GreenBalls is an es_list<circle>.

```
MyState1.addMisc( friction<circle> (&GreenBalls, .97) );
```

Shapes that use shapePhys already have a friction value stored in them, where RedBalls is an es_list<shapePhys<circle> >.

```
MyState1.addMisc( friction<shapePhys<circle> >(&RedBalls) );
```

gravity

To cause your objects to be affected by gravity, use the gravity class.
There are two ways to use it. First, with any shape, you specify a value as the second parameter. Second, if it is a shapePhys then a friction value is already stored within shapePhys object. A positive value will make an object float, a negative value will make it sink.

With any shape, where GreenBalls is an es_list<circle>.

```
MyState1.addMisc( gravity<circle>(&GreenBalls, -.05) );
```

Shapes that use shapePhys already have a friction value stored in them, where RedBalls is an es_list<shapePhys<circle> >.

```
MyState1.addMisc( gravity<shapePhys<circle> >(&RedBalls) );
```

wrapAround

The wrap around class will find shapes that go off screen, and wrap them to the other side.

This object will wrap around when it's location is beyond minimum X value of 0, min Y of 0, max X of 80, or max Y of 60.

```
MyState1.addMisc( wrapAround<circle>(&AsteroidRocks, 0, 0, 80, 60) );
// AsteroidRocks is an es_list<circle>
```

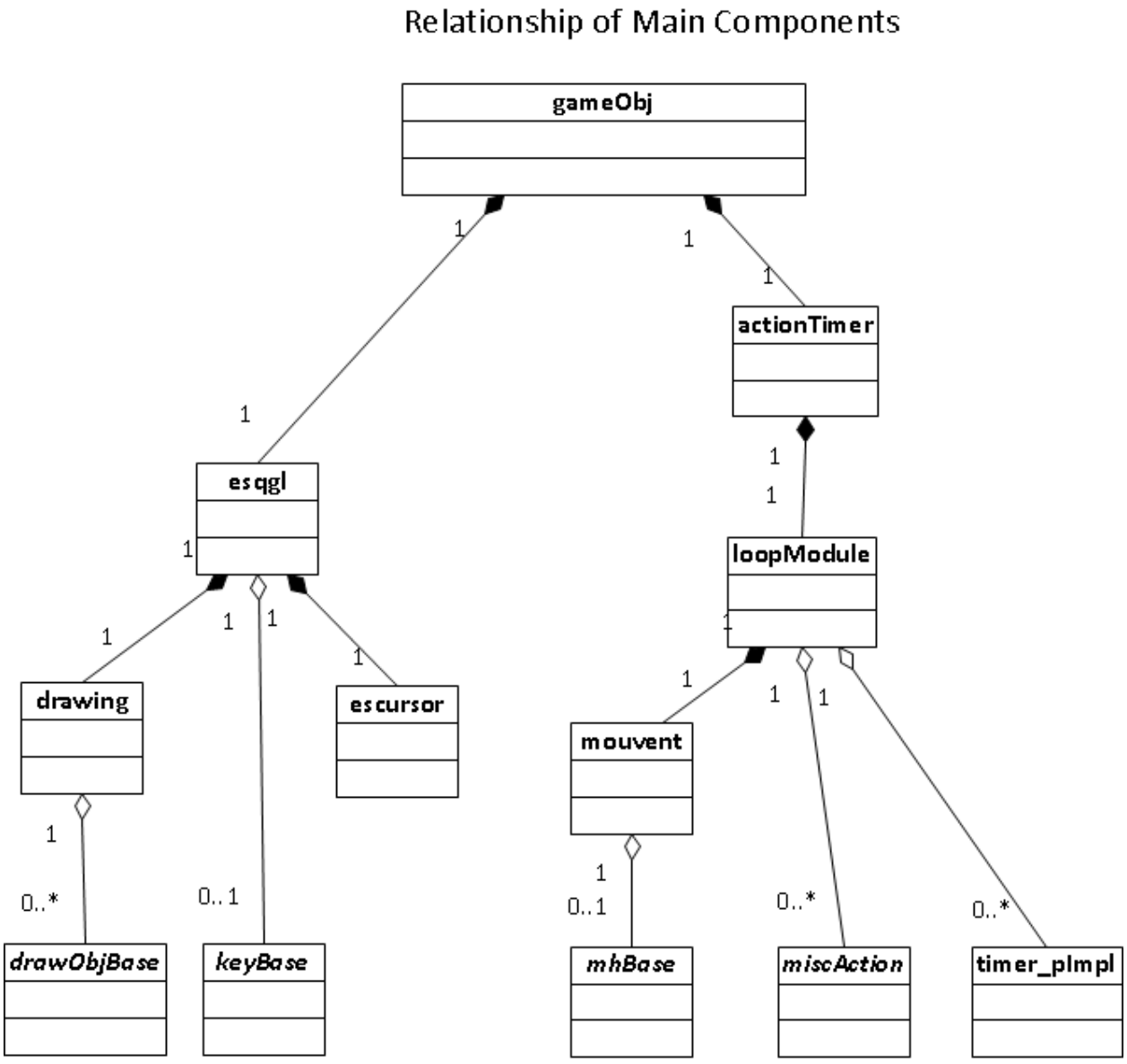
ang

The ang class can be used to keep track and manipulate angles. Here are a few examples of using the ang class.

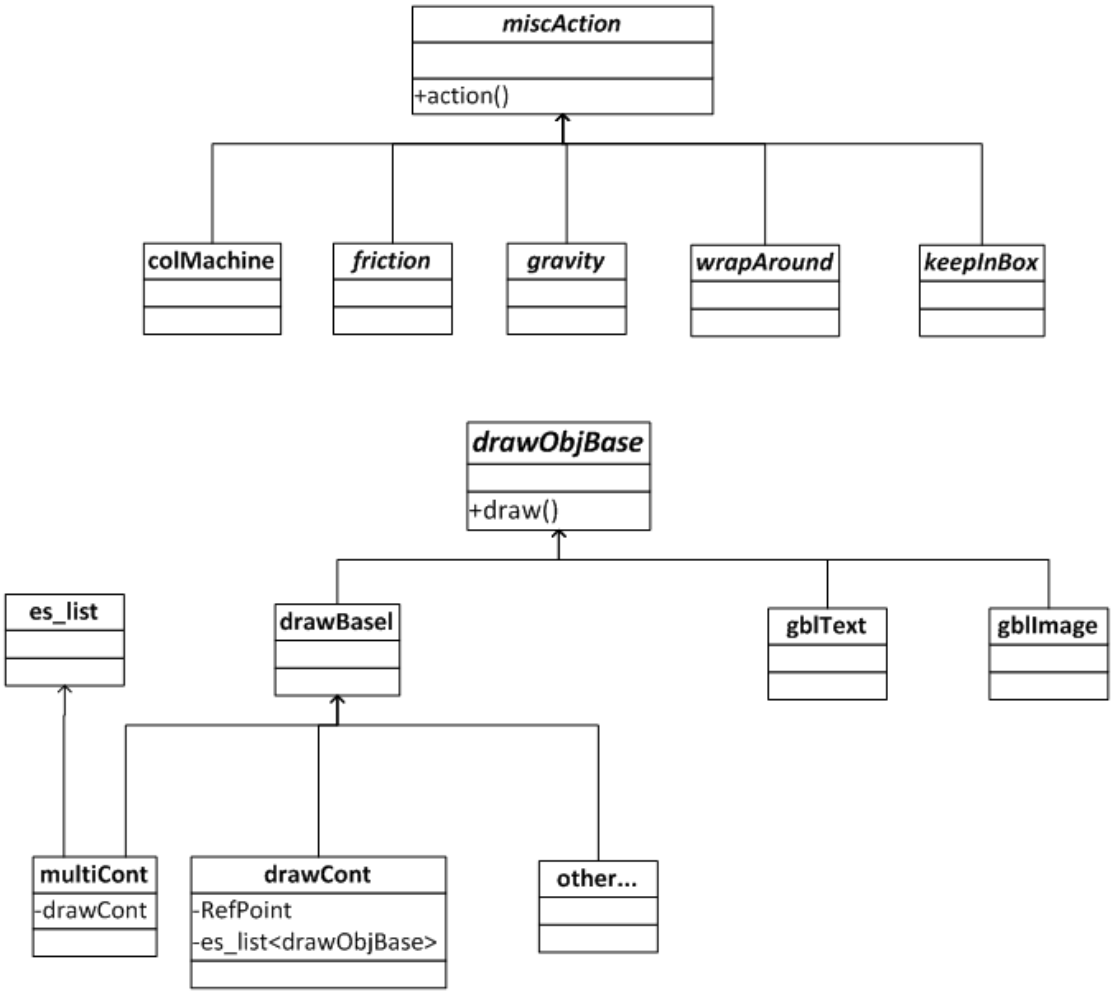
```
// Create using a radian value between negative pi and positive pi
ang Dir1 ( -.2 ); // creates angle with radian value -.2

// Create using one point
point2f  MyPoint( 0,1 );
ang Dir2 ( MyPoint ); // this will be equal to 90 degrees

// Create using two points
```

```
point2f p1(0,2);
point2f p2(1,3);
ang Dir3 (p1, p2); // This will be equal to 45 degrees

// Get value in degrees
cout<<"Dir3 == "<<Dir3.degrees()<<" degrees"<<endl;

// Many operators are overloaded for this class
Dir2 += Dir3; // Dir2 now should equal 135 degrees

// Get value in radians
cout<<"Dir3 == "<<Dir3.radians()<<" radians"<<endl;
// or
double cosOfAng = std::cos( Dir3() );

// See if angle is within two others, order of parameters is important
// Angle to test true is clockwise from first angle and ending at second angle
bool isWithinAngle =  Dir3.angleWithin( Dir1, Dir2 );

// See if angle is within two others, but always use the shortest angle
// parameter order not important, will use the shortest angle created
bool isWithinAngle =  Dir3.angleWithinShort( Dir1, Dir2 );

// Use the angle to calculate a point at a certain distance from 0,0.
point2f MyPoint = Dir3.getPointAt( 5 );
```

Class Diagram



Inheritance Relationships



Simple Example Program

Topics:

Basic Setup
Drawing Shapes
Collision System
Keyboard Events

```
int main(int argc, char *argv[])
{
gameObj MyGameObj(argc,argv);

gameComp MyGC;
MyGC.setRefrate(30);          // default: 30
MyGC.setWindim(800, 800);  // default: 800,600
MyGC.setWinpos(50,50);       // default:
MyGC.setScale(100,100);       // default: 80,60
```

An instance of gameObj needs to be created with every program that uses keyboard, mouse, window and drawing systems in GBL. Event handlers and other objects are added to the instance of gameObj.

The gameComp class simply stores basic information about the game and game window. The refresh rate (measured in milliseconds), window dimensions (in pixels), window location (in pixels), and InterXY(decimal values). InterXY is used to specify what x and y coordinates you want to work with. In this example, the window is 800 x 800 pixels, but the working space is 100 x 100. Mouse events and objects that are drawn will use the working space coordinates. So, when you draw a line from point (0,0) to point (100,0), the line will be drawn from the left bottom corner of the window to the right bottom corner.

```
es_list<dwall2f> Border;
Border.push_back(new dwall2f(point2f(0,0),point2f(100,0) ) );
Border.push_back(new dwall2f(point2f(100,0),point2f(100,100) ) );
Border.push_back(new dwall2f(point2f(100,100),point2f(0,100) ) );
Border.push_back(new dwall2f(point2f(0,100),point2f(0,0) ) );

es_list<circle> RedBalls;
RedBalls.push_back(new circle(5,point2f(10,10),point2f(0,0) ) );
RedBalls.push_back(new circle(5,point2f(90,10),point2f(0,0) ) );
RedBalls.push_back(new circle(5,point2f(90,90),point2f(0,0) ) );
RedBalls.push_back(new circle(5,point2f(10,90),point2f(0,0) ) );

es_list<circle> GreenBalls;
GreenBalls.push_back(new circle(5,point2f(50,10),point2f(0,0) ) );
GreenBalls.push_back(new circle(5,point2f(50,90),point2f(0,0) ) );
GreenBalls.push_back(new circle(5,point2f(10,50),point2f(0,0) ) );
GreenBalls.push_back(new circle(5,point2f(90,50),point2f(0,0) ) );

es_list<circle> YellowBalls;
YellowBalls.push_back(new circle(5,point2f(50,50),point2f(0,0)));
```

The above code creates for containers, or linked lists, using the class es_list (defined in listPack.h". The first list is a list of the type dwall2f, these are lines. The constructor of dwall2f takes 2 points, one for each side of the line. There are four lines added that make up the border. Next, there are four circles added to the RedBalls list, and four added to the GreenBalls list. The constructor's first argument is the circles radius, the second is the location, and the third is the "shift", or velocity, as an (x,y) value. Finally, we add one yellow circle.

```
MyGameObj.addToScene( drawFactory::drawCirc(&RedBalls, color::red) );
MyGameObj. addToScene( drawFactory::drawCirc(&GreenBalls, color::dgreen) );
MyGameObj. addToScene( drawFactory::drawCirc(&YellowBalls, color::yellow) );
```

Now we need to tell the drawing system how to draw our circles. The call to MyGameObj.addToScene() is used to add a draw object. Create this by calling the drawCirc() function, which takes two arguments, the list of circles and the color.

```
MyGameObj.addMisc( new friction<circle>(&RedBalls,.97) );
MyGameObj.addMisc( new friction<circle>(&GreenBalls,.97) );
MyGameObj.addMisc( new friction<circle>(&YellowBalls,.97) );
```

To add friction to the balls, we create a friction<circle> object. The constructor takes a pointer to the list of circles, and a .97. A 1 instead would apply no friction, a 0 would instantly stop the object. A .97 makes the circle will gradually slow down. A pointer to the friction object is passed to MyGameObj.addMisc() function.

Collision System

```
colMachine MyColMachine;
```

Now we want things to collide. First we create an instance of colMachine then add "gears" to it. A gear just tells the system what we want to collide, and what should happen to it.

```
MyColMachine.addGear( gearFactory::makeGear( Noth(&Border), Bounce(&RedBalls) ) );
MyColMachine.addGear( gearFactory::makeGear( Noth(&Border), Bounce(&GreenBalls) ) );
MyColMachine.addGear( gearFactory::makeGear( Noth(&Border), Bounce(&YellowBalls) ) );
```

To make sure none of the balls get past the border, add a gear connecting Border list with each list of colored balls. Each gear here says when a collision between a ball and a border occur, nothing should happen to the border and the ball should bounce.

```
MyColMachine.addGear(gearFactory::makeGear(Bounce(&GreenBalls), Bounce(&RedBalls)));
MyColMachine.addGear(gearFactory::makeGear(Bounce(&GreenBalls), Bounce(&YellowBalls)));
MyColMachine.addGear(gearFactory::makeGear(Bounce(&RedBalls), Bounce(&YellowBalls)));

MyColMachine.addGear(gearFactory::makeGear( Bounce(&GreenBalls)));
MyColMachine.addGear(gearFactory::makeGear( Bounce(&RedBalls)));
```

These calls will make sure all balls will bounce off of each other. The last two calls make green balls bounce off each other and red balls bounce off each other. Since there is only one yellow ball in the YellowBalls list, there is no need to add a gear for them to bounce off each other.

```
MyGameObj.addMisc( &MyColMachine);
```

The last thing we need to do for the collision system is to add the colMachine to the gameObj instance.

Keyboard Event Handling

```
class myKey : public keyBase{
   public:
        void keypress(esKeyEvent keyEvent){
          switch( keyEvent.getKey() )
          {
          case 'w' :
                    YellowBalls->front().shift.y +=2;
                    break;
          case 's' :
                    YellowBalls->front().shift.y -=2;
                    break;
          case 'a' :
                    YellowBalls->front().shift.x -=2;
                    break;
          case 'd' :
                    YellowBalls->front().shift.x +=2;
                    break;
          }
        }
        void keyrelease(esKeyEvent keyEvent){ }
        myKey(es_list<circle> *yballs){YellowBalls=yballs;}
    private:
        es_list<circle> *YellowBalls;
};
```

To handle keyboard events, we create a class that inherits from **keyBase** and redefines a keypress and keyrelease functions**.** The class keyBase has two virtual functions, keypress and keyrelease. We will only use keypress events in this example, so we define a keypress(esKeyEvent myevent) function in the class. We call myevent.getKey() in a switch statement, and use chars to figure out which key was pressed. If w, s, a, or d characters are pressed, the yellow ball's velocity will change.

```
boost::shared_ptr<myKey> TestKey(new myKey(&YellowBalls));
MyGameObj.setKeyHandler(TestKey);
```

Now we create a shared pointer to our myKey class, the YellowBalls list is passed to myKey constructor so we can change the direction of the ball when a key is pressed. Then we set the gameObj KeyHandler to our class by calling MyGameObj.setKeyHandler(TestKey).

```
return MyGameObj.exec(MyGC);
}
```

Now to start the game loop and start receiving events, we call MyGameObj.exec(MyGC). The gameComp instance we created at the beginning of this tutorial is passed as an argument to gameObj's exec() function. This function will not return until the game exits.

Change/add velocity of yellow ball by pressing w,s,a, or d.

Class and Namespace Reference

Public Classes

ang
circle
colMachine
comp2Single
compSingle
drawBasel
drawCircFill
drawCont
drawLineLine
drawIs
drawPoly
drawRectFill
drawRectLine
dwall2f
escursor
esKeyEvent
es_list
friction<shapePhys<T1>>
friction<T1>
gameComp
gameInfo
gameObj
gblAudio
gblColor
gblImage
gblMusic
gblSound
gblText
gblTimer
gravity<shapePhys<T1>>
gravity<T1>
hudCircBar
hudHBar
hudNumberBar
hudShapeBar
hudVBar
keyBase
gblTimer
matact
meter
mhBase
miscAction
multiCont
multiHandle
namespace esKey
namespace drawFactory
namespace shapeUtil
point2f
point2i
point3f
point3i
polPoint
shapePhys
vect
wraparound

Private Classes

actionTimer
adjust
baseBehav1
baseBehav2
basicDrag
basicThrow
bouncebounce
bouncedel
bouncenoth
bounceSingle
camera
colBolt1
colBolt2
color
colTrip
compMulti
compMultiSingle
delbounce
deldel
delnoth
delSingle
deq
drawing
esqgl
esqglinfo
gameComp
gameInfo
gameObj
listEl
listIter
loopModule
mouvent
namespace bnc
namespace colBhv
namespace fastDraw
namespace listalg
nothbounce
nothdel
polyPoint
shapePhys
timer_plmpl
trackpair
voidPtrLst