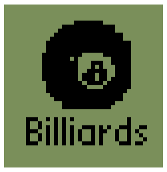
ELE101 PROJECT: BILLIARDS

The aim of this project is to write a billiards game for the Palm. This won't be a full-fledged game of eight ball, but rather a simpler game involving one ball to sink plus the cue ball. This project extends on some of the GUI concepts we have already learned in class by involving animated graphics and timing-dependent input. You will have the opportunity to practice some programming concepts that are exclusive to gaming applications, such as double-buffering of graphics and sprite transparency.

The complexity level of the project is only slightly higher than the last few assignments; however the amount of coding effort required is somewhat larger. We recommend you count on roughly twice the effort/time of a normal assignment.

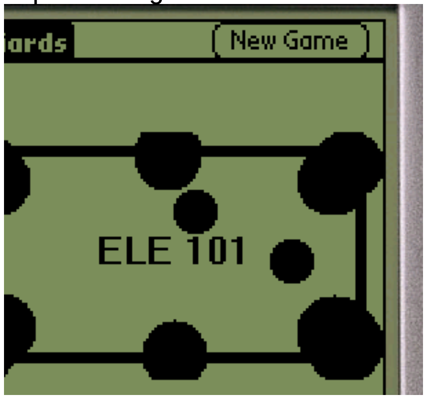
Your project submitted with the compiled PRC file and a zip file containing all source files *as well as an in-person one-on-one demo with the TA (James) are* ***due by May 11th, 2004.***



1. Getting Started

At first glance, the most daunting aspect of this project might be the physics involved. While we encourage the practice of using techniques from other fields to aid in programming, the purpose of this class is to learn basic programming so we've arranged this project in a way so that you will not have to spend much of your time reviewing physics concepts. Firstly, in our physical model we have made several assumptions that greatly simplify the equations. These concepts are equations are all presented for you in section 2 of this document. Secondly, all the source code for evaluating and solving these equations *has already been provided for you*. You will need to modify and expand upon these sections only in the case that you attempt some of the many extra-credit options.

A good way to get started would be to attempt to compile the shell that has been given to you and then run it in the emulator. This should give you an idea of the physics functions that have been made available to you, and you can also develop a sense for the missing features that you might go about implementing first.



2. Physics

Collision of pool balls involves momentum, kinetic energy, inelastic collisions, and rolling. Thanks to a number of simplifying assumptions, we'll be able to *ignore all four of the above concepts* and stick to just some basic ideas like velocity and acceleration. To begin with, we ignore rolling, so our billiards balls actually work more like flat round cylinders such as hockey pucks. Next, we assume that all of our balls (pucks) are the same mass, so our calculations will not involve momentum.

Before proceeding with some physics equations, let us lay down the mathematical foundation.

Vectors. A point in the xy plane can be represented by a vector with two components, an x-coordinate value and a y-coordinate value. We will use variable names in boldface to represent factors. For example, $\mathbf{v} = \langle x, y \rangle$ could represent the position of the center of one of our balls. Vectors will also be used to represent the distance and direction between two points, as well as velocities.

Magnitude of a vector. This is obtained using the Pythagorean theorem: $a^2 + b^2 = c^2$. In other words, $|\mathbf{v}|^2 = x^2 + y^2$ or $|\mathbf{v}| = \text{sqrt}(x^2 + y^2)$ where sqrt refers to the square-root function.

Dot products. To multiply two vectors and obtain a scalar value, we use the dot-product operator. The formula is simply $\mathbf{v}_1 \cdot \mathbf{v}_2 = x_1{*}x_2 + y_1{*}y_2$.

Projecting vectors. This one is a bit more complex, but you may have seen it before in classes such as linear algebra. Vectors can be separated into components, and to obtain a component of a vector in a certain direction we project that vector onto another vector. The formula for this is:
$\text{proj}_{\mathbf{u}}\mathbf{v} = (\mathbf{v} \cdot \mathbf{u})\ \mathbf{u} / (|\mathbf{u}|^2)$, where $\text{proj}_{\mathbf{u}}\mathbf{v}$ is read as the projection of **v** onto **u**. We can also write this formula as $\text{proj}_{\mathbf{u}}\mathbf{v} = (\mathbf{v} \cdot \mathbf{u})\ \mathbf{u} / (\mathbf{u} \cdot \mathbf{u})$, which is what we will use in this assignment. We use this operation in order to do things such as determine the amount of velocity transferred from a side collision. This operation helps us avoid more complicated or computationally expense methods such as angles and trigonometric functions (sines, cosines, etc.). Trigonometric functions, e.g. sines and cosines, are not necessary for doing any part of this assignment, even the extra credit options. However, many students may find it easier to reason and program some concepts use trigonometry. In this case, there are no such mathematical functions provided by our standard Palm OS library, but you have the option of using some special routines that we have provided in **mymath.c**. Because rapid calculation performance is essential for this assignment, these functions have been designed to execute quite rapidly. The cost is that the accuracy of their results is limited to within an error of +/- 0.001, but this error should be insignificant for the calculations necessary in this assignment.

Code for implementing these vector operations has been provided for you in the files **mymath.c** and **mymath.h**. Note that our math is done in all floating point so that you will not have to run into the complications of integer rounding. Next, we'll go over the physics involved and name the provided functions that take care of these operations.

Velocity. Velocity is the change in position with respect to time, so if we represent velocity with a vector v and position with a vector **d**, then $\mathbf{d}_{\text{final}} = \mathbf{d}_{\text{initial}} + \mathbf{v}t$

Friction. Kinetic friction is generally considered constant when the normal force (weight) of the object is unchanging. The rolling of billiards balls actually involves rolling friction, which in some ways is different from regular kinetic friction, but is represented by a formula of the same form. Since our acceleration (deceleration) due to friction is constant, we need not deal with the coefficient of kinetic friction, and instead use a deceleration parameter directly. To relate velocity and acceleration, our formula is
$\mathbf{v}_{\text{final}} = \mathbf{v}_{\text{initial}} + \mathbf{a}t$. Both the velocity and friction elements are handled by the following function:

```
ball_t MoveAndDecelerate(ball_t ball, float t);
    /* Move the ball one step along its path, and also slow the ball
       due to friction. Takes in a ball structure and a length of time
       in seconds for arguments. Returns a new ball structure. */
```

Detecting collisions. To do this we use a useful property of the dot product operation. If the dot product returns a positive value, we know that the angle between the vectors is less than 90 degrees. This gives us a criterion for determining if one ball is colliding with another ball. We have a vector representing the distance between the colliding ball and the stationary ball, and a positive dot product is sufficient criterion to tell us that there is a collision that we need to process.

```
Boolean DetectCollision(ball_t moving, ball_t stationary);
    /* Determines whether the first ball is colliding with the second ball. */
```

Collision velocity transfer. If two objects are of exactly the same mass and the first one collides head-on with the second, all of the motion will be transferred to the second object and the first one will be left behind *completely still*. In a real game of billiards you can observe this by hitting the cue ball directly into another ball (without spin). When the collision is not head-on, the result is not as simple, but we can actually still make use of the simplified result. Remember that our velocity vector can be separated into components however we like, and one way to do this is by separating the velocity vector into the component that would hit head on and the remaining component that goes directly off to the side. As we learned from the head-on collision case, 100% of the head-on component should be transferred to the target ball. Likewise, 100% of the non-head-on component would stay with the original moving ball. In a game of pool you can observe that when you normally hit another ball with the cue ball, the cue ball and the knocked ball split off *at right angles* with respect to each other.

```
vector_t Collide(ball_t moving, ball_t stationary);
    /* Returns a vector containing the velocity to transfer from the
       moving ball to the stationary ball. */
```

Superposition. In the last paragraph we covered what to do when a moving ball hits a stationary ball, but it seems things would get much more complicated if both balls were moving. Thanks to superposition, however, we can frame the complicated case as just the sum of two simple cases. The first ball can view the second ball as stationary when we calculate the result of that collision, and the second ball can view the first ball as stationary when we calculate the second result. The net resulting velocity transfer is just the sum of the two.

You can quickly see a list of all the function prototypes for these operations in **physics.h** and the corresponding implementations can be found in **physics.c**.

3. Graphics

In assignment 5 you were first exposed to some of the complications of painting the screen. When moving an object from one location on the screen to another, care had to be taken to ensure that garbage was not left behind in the object's previous location. One fairly simple way to do this was just to erase the whole screen and repaint the whole layout after every change. A side effect of this easy fix, however, is unnecessary flickering that happens because there must always be a point in time (after the erase) when parts of the screen will be blank.

While it is possible to work with the basic solutions of doing careful localized erasing and repainting on every graphics change, game programmers generally use a different much more robust solution: *double-buffering*. The idea is that instead of painting objects directly to the screen, the whole visual layout is first built up on another layer that is invisible to the user. When it comes times to display the next frame to the user, the entire picture on the hidden layer is copied to the visible layer. On desktops with standard video cards, specialized hardware is often used to perform a literal swap of the two layers. This completely eliminates flicker in a manageable way.

You are required to enable double-buffering for the graphics in this assignment. The program shell given to you has code that sets up an alternate window for painting, but it is your job to modify the framework so that this backbuffer is used for painting in order to eliminate flicker. A handle to the hidden window is saved in the variable OffscreenH. To use the buffer, you will need the following two functions:

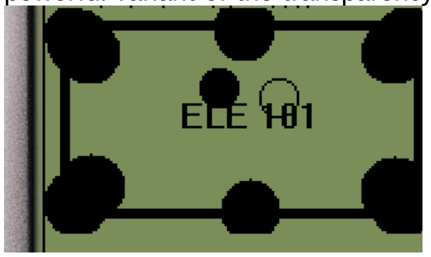
```
WinHandle WinSetDrawWindow(WinHandle winHandle);
```
Purpose: Set the draw window (All operations are relative to the draw window), and returns the original draw window. After you execute this function, all `WinDraw*` and `WinErase*` functions will affect only the new target window.

```
void WinCopyRectangle (WinHandle srcWin, WinHandle dstWin, const RectangleType
                    *srcRect, Coord destX, Coord destY, WinDrawOperation mode);
```
Purpose: Copy a rectangular region from one place to another (either between windows or within a single window.)
This function copies picture data between two windows, regardless of which one is the current display window. The rectangle that you provide to the function could be the one stored in the `displayrect` global variable. The final parameter (`mode`) in our case will be `winPaint`.

Aside from unnecessary flicker, the graphics in the shell program are quite bland. `WinDrawRect()` has been used to display each billiard ball. (The rounding of corners allows this function to display a “rectangle” as a circle.) However, we would like you to replace this and instead use a drawn image for each ball. Two such images are stored in the files **clearball.bmp** and **darkball.bmp**. Your task is to modify your program as well as your resource script in order to use these images. In order to load images C code, you will need to use the functions such as `DmGetResource()` to load each bitmap (image) and `MemHandleLock()` in order to reserve pointers to the bitmaps in memory. As an example, there is already code that takes care of this for **table.bmp** (the background image):

```
      backgroundbitmapH = DmGetResource('Tbmp', tableimage);
      backgroundbitmapP = (BitmapType*) MemHandleLock(backgroundbitmapH);
```

A second major element of game graphics programming is transparency. Normally we would draw rectangular bitmaps using the function `WinDrawBitmap()`, which is what has been done with the background bitmap. As for the billiard ball images (**clearball.bmp** and **darkball.bmp**), however, once you have managed to load these bitmaps you should display them with calls to `WinDrawBitmapTransparent()`, a function provided in **billiards.c**. To see the difference between drawing with transparency and without transparency, try displaying your images with `WinDrawBitmap()` and then compare the result to that when drawn with `WinDrawBitmapTransparent()`. On a side note, nowadays *alpha*-transparency (a much more powerful variant of the transparency we are using here) is used often in graphical applications.



Lastly, remember that if you'd like improve on any of the bitmap images, you can! Just edit them with a program like Microsoft Paint (MSpaint).

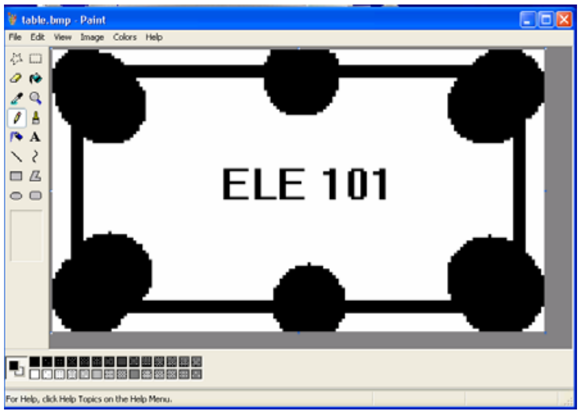


table.bmp - Paint
File Edit View Image Colors Help
ELE 101
For Help, click Help Topics on the Help Menu.

4. User input

To strike the cue ball in this game, the user is to use the stylus to make a motion across the screen and hit the ball. As was done in the puzzle project, you will need to handle `penDownEvent`. What makes this a little harder is that you will also need to handle `penMoveEvent` (stylus motion once it is pressed against the screen) and that timing is important so you will need to record the results of `TimGetTicks()` to use in your calculations. The results are to be passed to the `Strike()` function, which takes in a specified path and amount of velocity to transfer. Keep in mind that in reality a pool stick is more massive than the individual balls, so the velocity that you pass to `Strike()` can be scaled up by a factor to account for this.

```
vector_t Strike(vector_t start_spot, vector_t end_spot,
                vector_t transfer_velocity, vector_t ball_center) {
    /* For arguments this function takes a path of the cue stick and
     * the velocity we would want to add to the ball if we make a direct hit.
     * Returns the net velocity to be added to the target ball, or the zero
     * vector if we miss the ball. */
```

5. Game operation

Remember that we are implementing a full-fledged game that is to challenge the user somehow. The rules are to be as follows, and it is your task to implement a game engine as described below:
- The user begins with the cue ball and one dark ball (target ball) somewhere on the table.
- The user may perform a stroke to set the cue ball in motion. The user cannot perform another stroke until all balls on the table have settled down (motionless).
- Once the target ball is sunk into a pocket and the cue ball slows down and stops, a new target ball is placed randomly somewhere on the table. If the user accidentally sinks the cue ball, it is also placed back somewhere randomly on the table. To finish this part you would need to implement code that detects when a ball actually falls in the hole. For this you can use the `SubtractVectors()` and `MagnitudeVector()` functions to determine the distance between a ball and the center of a hole.
- The user may perform as many strokes as desired but the clock is always ticking, and the user has only ~~30~~ 300 seconds to sink all eight balls. In the top right corner of the screen you should display the number of seconds remaining. When the game is over the user can no longer knock any billiards balls around, but remember to make the "New game" button always operational.
- Scores is awarded to the user as follows: +100 points per target ball that reaches a hole, -100 points for accidentally putting the cue ball in the hole, and ~~+10 points~~ +1 point for every spare second left upon the completion of a round. On screen, we should see a display showing the user's score and time remaining.

Note: The timing/points-per-second-remaining requirements were changed so that you'd have an easier time testing out and demonstrating your program by sinking all eight balls. You are now free to set the clock limit at anything ranging from 30 to 300 seconds and the points awarded per remaining second to any number ranging from 1 to 10 points. Lastly, if you find the game to bee too easy under these rules, you are encouraged to mix things up and add more features to make it challenging.

6. Sound

For billiard ball collisions and when a ball falls in a hole, you need to add appropriate sound effects. For this project it is sufficient to use basic sound effects that can be generated using the function `SndPlaySystemSound()`. For example, the statement:

```
SndPlaySystemSound(sndClick);
```

produces a reasonable sound for billiard ball collisions.

7. Other possible enhancements

Multiple balls. Using superposition (section 2), you can expand this program to allow two or more balls plus the cue ball on the table at once. If you implement this enhancement, you are free to decide on what distribution the eight balls will appear on the table.

A moving sprite. A *sprite* is what the gaming industry calls any typical single animated object in a video game. An interesting addition to this billiards game would be to create a sprite that walks around on the table continuously even as the user is taking his or her shot. You can draw up your own images for this sprite using MSpaint, and can model it after Super Mario or any kind of figure, but it should have at least two distinct frames. Upon being hit by a billiard ball, the sprite should disappear, make a sound, and award the user with 300 points.

Transparency masks. Notice that our cue ball is was called the *clear* ball when it really ought to be the white ball. The reason we don't yet have a white ball is that the given `WinDrawBitmapTransparent()` has limited functionality. We cannot distinguish between the sections of the bitmap that we want to be transparent (the white space outside the circle) and the sections that should be opaque (the white space inside the circle). In gaming applications, a flexible transparency method is sometimes achieved by using *masks*. The use of masks involves a spare bitmap of identical size that specifies which pixels are to be the transparent ones. Your task for this enhancement would be to modify the `WinDrawBitmapTransparent()` routine so that it takes in an additional mask bitmap parameter and draws the bitmap as described.

Recording the top score. Palm databases are an important area that we've focused on near the end of the course, and this topic is certainly fair game for the final exam. This billiards project does not inherently involve databases, but as a good way to get some programming practice with this concept, consider adding in a feature for recording the highest score. The top score should be stored in a Palm database so that it can be saved even when the user exits and restarts your program, and you should display the top score alongside the user's current score. For some more database practice, you may wish to try something more advanced and store the top *five* scores in a sorted multi-entry database. For a multi-entry top score list, the list should be viewable by the user through clicking a menu.

Music. Before computers, video-game consoles, and arcade machines could easily output CD-quality audio, games implemented their music using MIDI (Musical Instrument Digital Interface). The Palm API has various MIDI routines you can use to generate music. As a challenge you could use these to add music to your billiards game, although it might require doing plenty of research on MIDI and how to use these advanced API calls.

The number of extra points awarded for each extra credit option has not yet been finalized, but as a guideline the more challenging feats will receive more points. If you have an idea for any cooler enhancement, then be sure to discuss it with the TA. It is definitely possible and encouraged to be creative with this assignment.

8. Summary of requirements & grading scheme

When doing this project, you can lay down the requirements in a checklist. Remember that many of these parts can be coded and tested independently, so you are encouraged to take things step by step, rather than attempting to code all features and doing single crash debugging session at the end.

A serial number and checksum are not required for this project, since you'll have to demonstrate your project on a real Palm in person anyway.

15 points:	Properly implement graphics double-buffering.
10 points:	Use images for billiards balls.
20 points:	Use stylus pen events to strike cue ball.
10 points:	Able to sink balls in holes.
10 points:	Sound effects.
10 points:	Game timer.
15 points:	Game operation and scoring.
10 points:	Good code style, comments, etc.
Extra credit (up to 30 points):	Explained in section 7.

Good luck! Have fun!