Cross-Platform Development with the Ecere SDK

As the founder of the **Ecere** open-source software project, I am pleased to share with you an introduction on how to build **native** cross-platform applications with the Ecere SDK. At the moment of writing, applications built with the SDK will work on **Windows** and **Linux**. It is also possible to build the SDK and the applications on other platforms, such as Mac OS X and FreeBSD, but there are still some minor issues to be resolved. Mobile platforms such as Android and iPad/iPhone are also targets we hope to support in the near future. The general idea is that you write your application once, with no particular attention to platforms, and then the exact same source code can be compiled and deployed for all supported platforms.

Overview of the Ecere SDK

Licensing

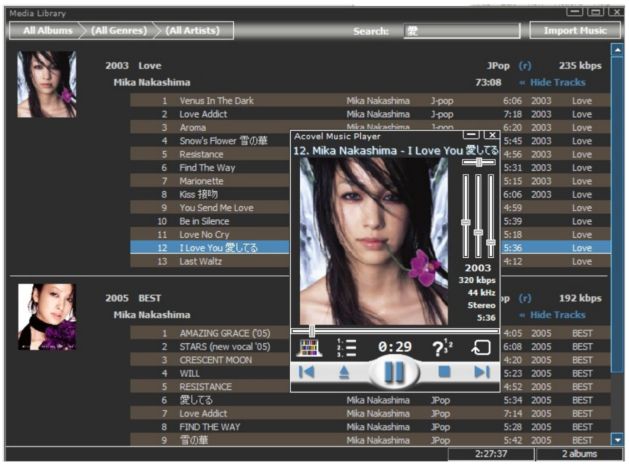
First, let me re-iterate that the Ecere SDK is **Free Open Source Software**, licensed under the **New BSD license**. This license is very permissive, in that the only condition to use the Ecere SDK in your applications is to make available the copyright and list of conditions within either the documentation (if released as binary) or source code (if released as source code). This means that, unlike software licensed under the GPL for example, it **can be used by applications which are not themselves open source.** Moreover, all third-party dependencies of the Ecere libraries are released under similar terms, which would otherwise make this impossible.

What's included

- A set of compiling tools for the **eC** language (*see next section about eC*)
- An **Integrated Development Environment**, with the usual features such as:
  - A source **code editor** with auto-completion, syntax highlighting
  - Management of application and library **projects**
  - A visual **debugger**
  - A **Rapid Application Development** form designer, based on properties & methods

- A run time library, providing a **uniform AP**I across platforms, featuring:
  - A **GUI** toolkit (with a vast collection of powerful controls: Buttons, Edit boxes, Drop/Combo boxes, Menus, Tabs, Tree views/Grids/List boxes, file dialogs, ...)
  - A **2D** graphics API (**bitmaps, fonts, international text, basic drawing)**
  - A **3D** graphics API, supporting both **Direct3D** and **OpenGL (3DS file format support)**
  - A networking API which provide **Sockets** as well as a **distributed objects** system for eC
  - System functionality such as file access, multi-threading & synchronization, handling date & time, etc.

- Additional libraries and code for more features, such as:
    - The Ecere Data Access (**EDA**) layer, an abstract relational database API, providing an active record system for eC. Currently it has drivers for a minimal Ecere RDBMS and **SQLite** (as well as an encrypted version using SQLiteCipher), and recently a basic **Oracle** driver was introduced
    - An audio library (supporting **DirectSound** on Windows and **ALSA** on Linux)
    - **WIA** Scanning support on Windows
    - **SSL** Sockets suport through **OpenSSL**
    - A 2D **tiled based** game graphics engine (Tiled map, Sprites, A*)
- A collection of sample applications showcasing how to use the Ecere SDK



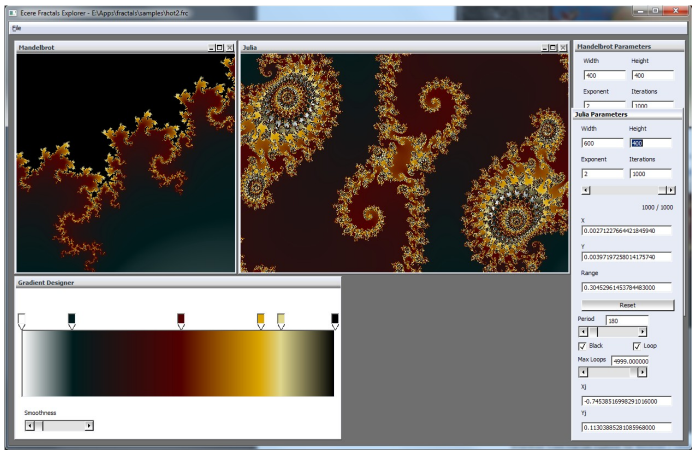
2003 Love JPop (r) 235 kbps
Mika Nakashima 73:08

1	Venus In The Dark	Mika Nakashima	J-pop	6:06	2003	Love
2	Love Addict	Mika Nakashima	J-pop	7:18	2003	Love
3	Aroma	Mika Nakashima	J-pop	6:20	2003	Love
4	Snow's Flower 雪の華			5:45	2003	Love
5	Resistance			4:56	2003	Love
6	Find The Way			5:31	2003	Love
7	Marionette			5:15	2003	Love
8	Kiss 接吻			6:06	2003	Love
9	You Send Me Love			4:59		Love
10	Be in Silence			5:39		Love
11	Love No Cry			5:18		Love
12	I Love You 愛してる			5:36		Love
13	Last Waltz			4:12		Love

2005 BEST p (r) 192 kbps
Mika Nakashima

1	AMAZING GRACE ('05)			4:05	2005	BEST
2	STARS (new vocal '05)			6:08	2005	BEST
3	CRESCENT MOON			4:20	2005	BEST
4	WILL			5:23	2005	BEST
5	RESISTANCE			4:52	2005	BEST
6	愛してる	Mika Nakashima	JPop	5:34	2005	BEST
7	Love Addict	Mika Nakashima	JPop	7:14	2005	BEST
8	FIND THE WAY	Mika Nakashima	JPop	5:28	2005	BEST
9	雪の華	Mika Nakashima	JPop	5:42	2005	BEST

2:27:37 2 albums



About eC

The Ecere SDK is implemented with and provides its API for the eC programming language.
eC is an object-oriented programming language based on C (it's a 'C with classes', like C++, C#, D, Java...). While maintaining all of C's syntax, features and functionality (such as compiling to native code, high portability and performance, access to system libraries, great level of interoperability and compatibility with existing libraries), eC provides modern features, including:

- Classes with inheritance and polymorphism
- Virtual methods at the instance level (a new class is not required to override a method)
- Object properties (set/get encapsulation)
- Reflection (Querying methods, members, properties, classes...)
- Importing mechanism (no need for header files)
- Dynamic module importing (Loading/unloading modules at runtime, useful for plugins)
- No need for prototypes (forward declaration)

In the future, we might provide APIs to develop with the Ecere SDK using other object oriented programming languages. We hope to allow interoperability between eC and those languages through an automatic bindings generation system.

The eC compiler is currently implemented by compiling to C as an intermediate language. The build system will then automatically invoke GCC to compile those intermediate C files to object files (this process is mostly transparent to the user when using the Ecere IDE or Makefile generation tool).

Through its properties, which enable assignments to dynamically reflect state changes, its simple instantiation notation *`Class object { member = value }`* and its object data types, eC provides an elegant syntax well suited for GUI applications.

Although the object instances are in fact pointers, they are not presented to the programmer as such, and so the confusion added by the extra reference level, the pointer notation (`Class *`) and the `->` used in C++ are avoided, keeping the simpler member access notation: *`object.member = value`*.

For the sake of example, here is an entire message box application written in eC with the Ecere toolkit:

```
import "ecere"
MessageBox msgBox { caption = "Title", contents = "hello, world!!" };
```

Obtaining and installing the Ecere SDK

The home of the Ecere SDK on the web is at http://ecere.com .

There you will find both binary and source distributions of the SDK, as well as links to our support forums, bug trackers, and other useful resources. On the front page, you will find platform icons which will bring you to the corresponding sections of our Download page.

Windows

If you click the Windows icon, you will find our binary installer for the latest release, as well as instructions regarding MinGW, should you chose to use your own installation of MinGW (A minimal system to run the GNU GCC compiler on Windows). If you use the full installer, the process should be quite straightforward, and you'll be able to simply click the Ecere IDE icon for a fully configured IDE to come up. If you use your own installation of MinGW, you'll have to make sure that it is properly installed and that the path to gcc.exe and mingw32-make.exe are in your PATH environment variable, or you can alternatively configure the paths in the File → Global Settings dialog of the IDE, under Compilers → Directories → Executables.

Linux

If you click the GNU / Linux icon, you will find instructions on how to obtain Ecere for Ubuntu from the Ecere PPA archives, as well as down-loadable Debian packages. You will find there a list of package dependencies, as well as notes regarding problems you might encounter where text does not show up (either a missing font or outdated graphics driver issue). There are also links to ArchLinux packages, and other distributions, for which you will have to build the SDK from source.

The Mac icon will, sadly, bring you to notes on the currently experimental status of the SDK on the Mac, which at this point can only run through the X11 server and suffers some usability issues. We hope to change this soon.

Git

The Git icon will bring you to our home on GitHub, where we host code with the Git version control system. GitHub has a great front end to Git, and is a great place to collaborate. If you want to keep up with the code changes to Ecere or contribute, this is where it all happens. You can use Git to clone the sdk source code from here and simply issue a pull to download the latest incremental changes. The command to clone the SDK into a folder within the current working directory 'sdk' would be: **git clone git://github.com/ecere/sdk.git** . On Windows the **msys-git** implementation of git works great. You will find a link to the latest version from the Source section on our wiki's download section.

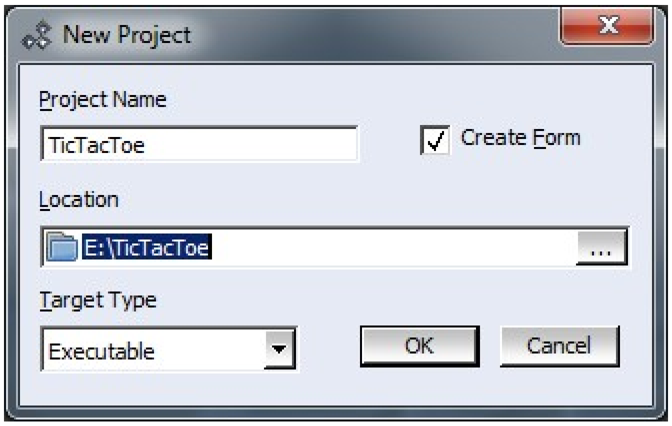
Source

Finally, if you don't feel like setting up a git clone, a big shiny Download button will download the very latest code as single tarball. Regardless of your platform, after you've cloned or extracted the source code you should be able to go to the sdk's top level directory and simply issue a 'make' command (mingw32-make on Windows). Hopefully everything will go well and you will see 'The Ecere SDK has been built successfully', at which point you can proceed to issue a 'sudo make install' on Linux, or a 'mingw32-make install' on Windows. For make install to work on Windows Vista/7, you will need an Elevated Command Prompt. You can be start one by clicking on the Start button, typing **cmd**, right clicking the cmd.exe icon selecting **Run as administrator,** and selecting **'Yes'** in the UAC prompt. If you have any issue with the build or installation, you're welcome to ask for help in the forums.

Setting up a new project with the IDE

Let's start! First, launch the Ecere IDE. To be able to build any application, we will require a project. Let's create a new project: using the menu bar's Project → New. We need to provide a location for our project, and a project name. Use a new folder for it, making sure you have the right permissions in the parent folder to create it. For the folder location, if you are on Windows, use a path containing only ASCII characters (MinGW-make does not seem to support Unicode paths properly). Always Stick to ASCII characters for the project name. We'll be making a TicTacToe game, so let's name our project `TicTacToe`. The IDE will create the folder if it does not exist.

Leave the '*Create Form*' check box ticked, as we will be making a GUI application (As opposed to a console based application, such as the typical `hello, world!!`). The target type specifies whether we want to build an executable application or a library to be used as a component part of another application. In our case we want to build an executable. After pressing OK, our project will be ready to use.



Project Name: TicTacToe
Create Form: ✓
Location: E:\TicTacToe
Target Type: Executable

You should now be seeing a blank form, with a default file name of "form1.ec". We would like to use a different file name, so we will change that. Press F8 (twice if the form was not active) to toggle back to the Code Editor (as opposed to the Form Designer). You should now be looking at the code for form1.ec. Select All (Ctrl-A), cut it into your clipboard (Ctrl-X), close it (Ctrl-F4 – twice to close the form designer as well, *No* for not saving), go to the project view (Alt-0), hit 'Delete' on *form1.ec* to take it out of the project. Now we'll add a file named *TicTacToe.ec* instead. Move up to the project node (TicTacToe.epj), and either right click on it and select *Add Files to Project*, or simply press **enter** while it is selected. Then type in the name of the file to add, *TicTacToe.ec* (it does not need to exist prior to adding it). Notice how the new file is added under the target node. Now double click on it and add back the initial form code from your clipboard:

```
import "ecere"

class Form1 : Window
{
   caption = "Form1";
   background = formColor;
   borderStyle = sizable;
   hasMaximize = true;
   hasMinimize = true;
   hasClose = true;
   size = { 576, 432 };
}

Form1 form1 {};
```

Now, try building the application. Select from the menu bar Project → Build (shortcut key – F7). If everything is configured correctly, you should get the following output in the build output tab:

```
Default Compiler
TicTacToe-Debug.Makefile - Project has been modified. Updating makefile for Debug
config...
Building project TicTacToe using the Debug configuration...
Generating symbols...
TicTacToe.ec
Compiling...
TicTacToe.ec
TicTacToe.c
Writing symbol loader...
TicTacToe.main.ec
TicTacToe.main.ec
TicTacToe.main.c
Linking...

TicTacToe (Debug) - no error, no warning
```

If you are not getting this, but errors instead, the Ecere SDK might not be installed properly. Please refer to the installation notes again. If you are getting syntax errors, you might not have pasted the code properly. Here is the unfortunate result of missing the last semicolon:

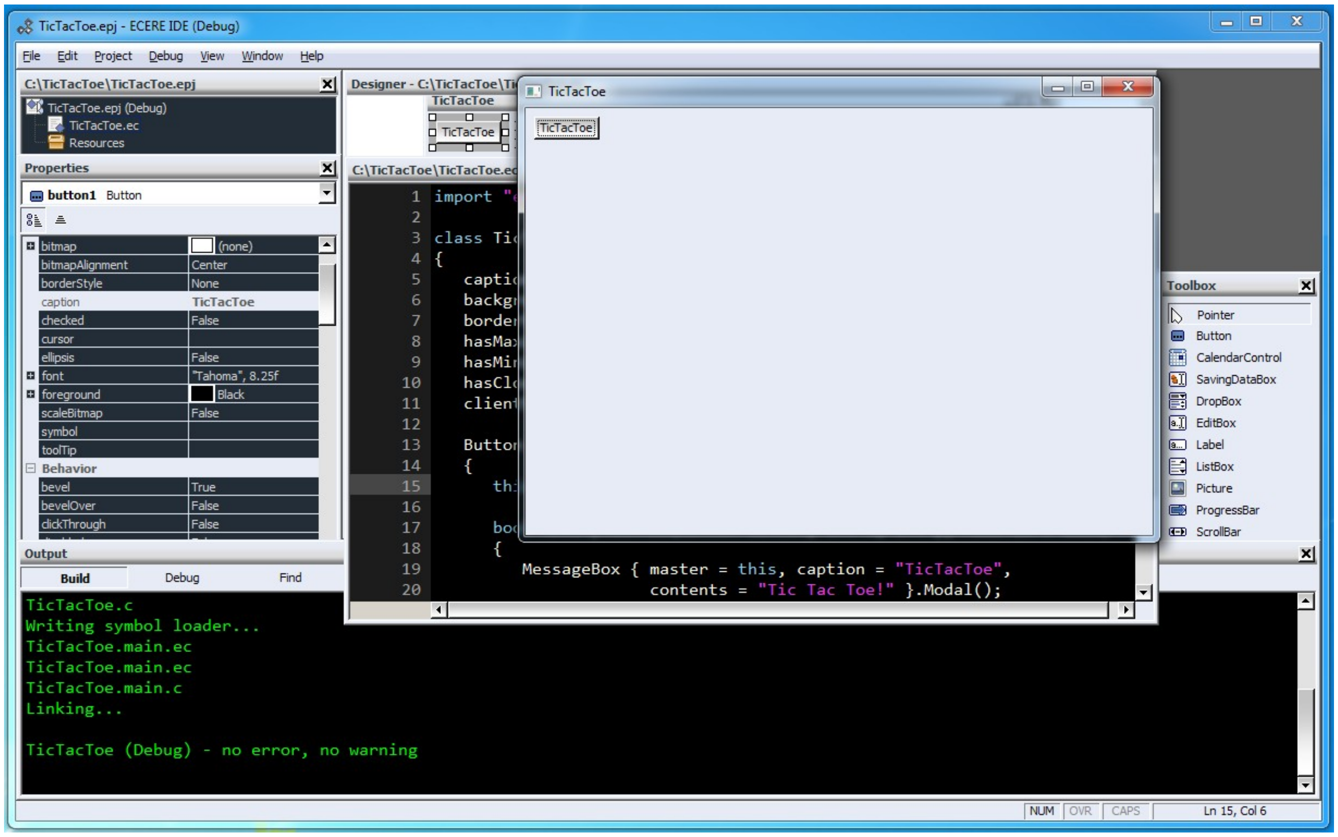
```
Compiling...
TicTacToe.ec
   TicTacToe.ec:15:1: error: syntax error

TicTacToe (Debug) - 1 error, no warning
```

Double clicking or hitting enter on the error line in the build output view will bring you directly to the offending line of code. If everything went right, you should now have built your first eC program. You can now try running with F5 (Debug → Start). You should see your blank form coming up; it can be closed either with the Close button or the **Alt-F4** keyboard shortcut.

Quick Introduction to eC and the Ecere GUI system

Now let's tweak it a bit. We'll change the name of the class from *Form1* to *TicTacToe*, and its instantiation at the bottom. We will name the class instance `mainForm` instead of `form1`. We will also change the caption of the window from "Form1" to "TicTacToe". All of the changes *within* the class (i.e. all changes except for those to the instantiation at the end) can be performed through the property sheet on the left side of the form designer (F4 to toggle between properties and methods). The code should now look like this:

```
import "ecere

class TicTacToe : Window
{
   caption = "TicTacToe";
   background = formColor;
   borderStyle = sizable;
   hasMaximize = true;
   hasMinimize = true;
   hasClose = true;
   size = { 576, 432 };
}

TicTacToe mainForm {};
```



bitmap	(none)
bitmapAlignment	Center
borderStyle	None
caption	TicTacToe
checked	False
cursor	
ellipsis	False
font	"Tahoma", 8.25f
foreground	Black
scaleBitmap	False
symbol	
toolTip	
Behavior	
bevel	True
bevelOver	False
clickThrough	False

```
import "
 
class Ti
{
   captio
   backgr
   border
   hasMax
   hasMin
   hasCl
   client
 
   Button
   {
      thi
 
      bo
      {
         MessageBox { master = this, caption = "TicTacToe",
                      contents = "Tic Tac Toe!" }.Modal();
```

```
TicTacToe.c
Writing symbol loader...
TicTacToe.main.ec
TicTacToe.main.ec
TicTacToe.main.c
Linking...

TicTacToe (Debug) - no error, no warning
```

Try to run the new code; notice your changes. Now let's try adding a button that will perform an action. Use F8 to switch back to the Form Designer, click the "Button" icon in the Toolbox at your right, and drag it (while holding left mouse button down) onto the form. You can try moving the button around on the form by click-and-dragging it. Double-clicking the button will automatically override the *NotifyClicked* event for the button, which is invoked whenever the user clicks the button. The code will now look like this:

```
import "ecere"

class TicTacToe : Window
{
   caption = "TicTacToe";
   background = formColor;
   borderStyle = sizable;
   hasMaximize = true;
   hasMinimize = true;
   hasClose = true;
   clientSize = { 576, 392 };

   Button button1
   {
      this, caption = "TicTacToe", position = { 8, 8 };

      bool NotifyClicked(Button button, int x, int y, Modifiers mods)
      {

         return true;
      }
   };
}

TicTacToe mainForm {};
```

The *`Button`* class, like all visible GUI components in the Ecere GUI, inherits from the base *`Window`* class. Our *`TicTacToe`* class also inherits from *`Window`*, as the *`class TicTacToe : Window`* stipulates, a syntax which will be familiar to programmers of most other 'C with classes' languages.

The *`this`* identifier, which you see within the instantiation of the *`Button`* object, refers to the current class (*`TicTacToe`*). It is being assigned to the '*`parent`*' property of the *`Button`* class, *`parent`* being the first initializable member of all classes deriving from *`Window`*. As another example, `x` and `y` are the first (and only) initializable members of the *`Point`* class expected for the *`position`* property where the code says: `position = { 8, 8 }`. Thus, the parent could alternatively be assigned as: `parent = this`.

The *parent* of a window in the Ecere GUI is the window within which it is confined. The parent of top level windows is the *desktop*, which is the default if no parent is specified (or if a value of *null* is specified).

Note that the property assignments directly within the class (e.g. *`caption`*, *`background`*, *`borderStyle`*, *`hasMaximize`*...) are default property values for the *`TicTacToe`* class (all instances of it), whereas the property assignments within the instantiation of the *`button1`* object are values assigned specifically to that particular instance of the *`Button`* class. Default values for a class can be overridden at the instance level, for example here we could override these values when instantiating *`mainForm`*.

Now within this *NotifyClicked* event, we will bring up a message box that says *Tic Tac Toe!*. To do so, we need to instantiate an object of the *MessageBox* class. Because the message box is temporary, it does not need to be named, so we'll use an *anonymous instance*. The syntax is very similar to the instantiation of our named *TicTacToe* class (the *mainForm* instance), without the name:

```
bool NotifyClicked(Button button, int x, int y, Modifiers mods)
{
   MessageBox { master = this, caption = "TicTacToe",
                contents = "Tic Tac Toe!" }.Modal();
   return true;
}
```

In eC, the curly braces are the instantiation operators, inspired from the declaration list initializers of C, taking out the assignment operator (`Vector3D vec = { 3, 4, 5 };` becomes `Vector3D vec { 3, 4, 5 }`). The use of curly braces for objects will also be familiar to those accustomed to the Java Script Object Notation (JSON). Whereas an anonymous instantiation is considered a *statement*, a named instantiation is considered a *declaration*. This is important to note, since eC requires all declarations to be grouped at the beginning of a compound block: no declaration can follow a statement within the same compound block. This follows the C89 convention. A syntax error will result if a declaration comes after a statement.

As in our early example, we will set properties for the message box: the *caption* (what shows up in the title bar), and the *contents* (the actual text that goes within the box). Both properties (that can perform actions) and data members (regular C structures data members) can be assigned within the curly braces. We will keep the default *type*, which is a message box with only an **OK** button.

To establish the relationship between the message box and our main form, we will set its *master* property to be the current instance of the *TicTacToe* class. This will state that the message box is owned by the main form. If no *master* is specified for a window, the default is that window's *parent*. The master for a control also determines who will receive the notification events. For example, in the case of our button, the *TicTacToe* class (the parent of the button, also the master since no master is specified) receives notification events for the window, so the TicTacToe class can be referred to as *this* within the NotifyClicked event. Setting the master of the MessageBox to be the main form will enable the message box to be modal in respect to the main form, as explained below.

In addition to instantiating the GUI object itself, whose purpose is to hold the associated data, the *MessageBox* (like any *Window*) must be explicitly created, unless it is auto-created by virtue of being a *global instance* or a *member instance* of another *Window* being created (such as the case of our *mainForm* instance of the *TicTacToe* class). This is normally done through the *Window::Create()* method, though here we will use the *Window::Modal()* method, which has the triple purpose of making the dialog **modal** (through the *isModal* property, no other related window will accept input until the message box is closed), **creating** the window and **waiting** for the window to be closed before returning from the method call. Run the program again it to see it in action.

For a more in depth coverage of the features of the eC programming language, please consult the *Ecere Tao of Programming*, a Programmer's Guide for the Ecere SDK (a work in progress). You will find the *Tao* installed along with the SDK (In *Program Files/Ecere SDK/doc* on Windows, */usr/share/ecere/doc/* on Linux), or online at http://ecere.com/tao.pdf. The first section of the *Tao* covers the C foundations of eC, whereas the second section goes over the object oriented concepts of eC.

For the rest of this article, we will focus on the functionality allowing us to build a *TicTacToe* game with the SDK (which, of course, can be compiled and deployed on any supported platform).

Drawing graphics with Ecere

The application model of Ecere is built around the classic game development main loop concept:

> ***While the application is running:***
>
> - Wait for input
> - Process input
> - Render current state

As such, the GUI system expects drawing to occur solely as part of this last *rendering* step. Any GUI component must therefore keep track of its current state, and any visual change is initiated by a two steps process:

1. Modify the state: usually done by modifying member variables of the GUI (*Window*) object
2. Request an update: a passive request to the GUI system to be updated on the Rendering phase of the next cycle. This is done by the `Window::Update()` method, with an optional parameter specifying the area to be updated (or *null* for the entire Window to be updated).

The drawing itself is handled in the `Window::OnRedraw` virtual method, called back by the GUI system during the rendering phase. The OnRedraw method receives a `Surface` in which to render the current state of the object. The `Surface` class provides the methods for rendering bitmaps, text (with support for various fonts and international text using UTF-8), as well as basic operations such as line drawing and area filling.

The following OnRedraw sample renders a simple blue, horizontal, 360 pixels wide by 10 pixels high, filled rectangle, at position (x = 20, y = 135) from the top-left corner of the window's *client area* (the portion of the window excluding its decorations such as the title bar, resizing bars, scroll bars):

```
void OnRedraw(Surface surface)
{
   surface.background = blue;
   surface.Area(20, 135, 379, 144);
}
```

Note that the `background` property affects the color of calls to `Area()`, whereas `foreground` affects the color of lines drawn with calls such as `Rectangle()`, `DrawLine()`, `HLine()`, `VLine()`, as well as the color of text rendered with methods such as `WriteText()`.

Now let's try to display a TicTacToe grid. First, we will tweak our TicTacToe form class definition to have a square shape, by settings the clientSize property to 400x400. We will also get rid of the resizable border, minimize and mazimize button, keeping only the close button (which automatically gives the window a `fixed` border on which to place the button, if no border style is specified). We will change the color of the form to pure white as well:

```
background = white;
hasClose = true;
clientSize = { 400, 400 };
```

When drawing the grid, we will base its dimensions on the window size, to make it easily adjustable by simply modifying the `clientSize` property of the class.

We will define some constants as well, at the top of the file, using eC's `define` mechanism:

```
define spacing = 20;
define lineWidth = 10;
```

As the first step of drawing our grid, we will compute how much space each of the 3 sections of the grid should take, evenly dividing by 3 the available space (after taking out the spacing at both ends), we will name these variables `sw` and `sh` for section width and height:

```
      int sw = (clientSize.w - 2*spacing) / 3;
      int sh = (clientSize.h - 2*spacing) / 3;
```

Our grid is then rendered with the following 4 calls to `Area()`:

```
      // Vertical lines
      surface.Area(spacing + sw   - lineWidth / 2,   spacing,
                   spacing + sw   + lineWidth / 2-1, clientSize.h - spacing - 1);
      surface.Area(spacing + sw*2 - lineWidth / 2,   spacing,
                   spacing + sw*2 + lineWidth / 2-1, clientSize.h - spacing - 1);

      // Horizontal lines
      surface.Area(spacing,                     spacing + sh   - lineWidth / 2,
                   clientSize.w - spacing – 1, spacing + sh   + lineWidth / 2-1);
      surface.Area(spacing,                     spacing + sh*2 - lineWidth / 2,
                   clientSize.w - spacing – 1, spacing + sh*2 + lineWidth / 2-1);
```

Try to put this together to see the grid (you can refer to the full listing of the *TicTacToe* game at the end of this article in case you get confused how things fit together).

Our next step is to keep track of the state of the game. For this purpose, we will use an enumeration type along with a 3x3 array:

```
enum TTTSquare { _, X, O };
TTTSquare board[3][3];
```

As a global object, the board will automatically be initialized with '0' values by default, which will match to the '_' (empty) value of our TTSquare enumeration type. For the purpose of our initial testing however, we will initialize it to some arbitrary state so we can make sure drawing X's and O's works:

```
TTTSquare board[3][3] =
{
   { _, X, O }
   { O, _, _ },
   { _, _, X }
};
```

Now let's write code to render the X's and O's. For the sake of simplicity, we will use text and fonts (we could have chosen to use bitmaps instead and use the `Surface::Blit()` method to display them). First, we will create a `FontResource` object to automatically load and unload our font when required. The Ecere graphics system supports dynamic display mode change, e.g. switching from Direct3D to OpenGL, or changing color depth while the application is running. This can be handled through `Window`'s `OnLoadGraphics` / `OnUnLoadGraphics` callback virtual methods, but the `FontResource` and `BitmapResource` classes provide automatic management of Fonts and Bitmaps:

```
FontResource tttFont { "Comic Sans MS", 50, bold = true, window = this };
```

Here we have selected "Comic Sans MS" for the *faceName* (the first property) of our font, a *size* of 50 font points and a *bold* weight.

By setting the *window* property of the *FontResource* to our *TicTacToe* instance, the font will automatically get loaded and unloaded for use within the *display system* of our window. By default, all windows of an application share the same display system, but with Ecere it is possible for one window to work with OpenGL while another runs in GDI or X11 mode, in which case multiple display systems are in use (and multiple fonts/bitmaps objects must be loaded). The *BitmapResource* class works in a very similar way to the *FontResource* (in fact they both inherit from a common *Resource* class).

The *FontResource* is purely a resource management object. The actual *Font* object to be used for rendering can be accessed through its *font* property, which can be set on a *Surface* as such:

```
surface.font = tttFont.font;
```

In order to center the X's and O's within the squares of the grid, it will be necessary to obtain the dimensions of each letter. To do so we will use the *Surface::TextExtent* method, after having selected our font:

```
int Xw, Xh, Ow, Oh;
surface.TextExtent("X", 1, &Xw, &Xh);
surface.TextExtent("O", 1, &Ow, &Oh);
```

The first parameter of *TextExtent* is the string to display, the second the length (only 1 character), followed by the addresses of 2 integer variables to retrieve both the width and height of the string.

We will then use *Surface::WriteText* to display the letters at the appropriate location, using the section width and height variables from earlier again (sw and sh) in our computations. The proper entry in our two-dimensional board table is examined to see whether nothing, a X, or a O is to be rendered. X's are displayed in green, whereas O's are displayed in red.

```
int x, y;
for(y = 0; y < 3; y++)
{
   for(x = 0; x < 3; x++)
   {
      TTTSquare p = board[y][x];
      if(p == X)
      {
         surface.foreground = green;
         surface.WriteText(spacing + sw * x + sw / 2 - Xw/2,
                           spacing + sh * y + sh / 2 - Xh/2, "X", 1);
      }
      else if(p == O)
      {
         surface.foreground = red;
         surface.WriteText(spacing + sw * x + sw / 2 - Ow/2,
                           spacing + sh * y + sh / 2 - Oh/2, "O", 1);
      }
   }
}
```

We have organized the whole task of rendering the X's and O's within the *DrawPieces* method of the TicTacToe class, which will be invoked from the *OnRedraw* method.

Processing Input

The Ecere GUI provides method callbacks to handle mouse and keyboard input within a Window. Keyboard events are received by the *OnKeyDown*, *OnKeyUp* and *OnKeyHit* methods. *OnKeyHit* is normally used for handling characters, which can be repeated while the key is held down. The input methods will relay the character information provided by input methods (IMEs), which can be composed by multiple key presses releases. *OnKeyUp*/*OnKeyDown* is normally used to perform action associated with a specific key. It is also possible to query the state of keys, which is most useful in the context of a video game.

For handling mouse input, nine callback virtual methods of the Window class can be overridden, three buttons times three types of events:

*On[**Left/Middle/Right**][**ButtonDown/ButtonUp/DoubleClick**]*.

Mouse wheel support is handled as special key values within the *OnKeyUp* or *OnKeyHit* method: ***wheelUp*** and ***wheelDown***.

For our small TicTacToe game, we will simply process *OnLeftButtonDown*:

```
bool OnLeftButtonDown(int mx, int my, Modifiers mods)
{

   return true;
}
```

Here we have modified the default parameters names from x and y to mx and my, because we wish to reserve x and y for the indices within our board table. The first thing we will check when the mouse button is pressed is whether we are within the TicTacToe grid, which is defined by the spacing and clientSize of our class:

```
if(mx >= spacing && mx < clientSize.w - spacing &&
   my >= spacing && my < clientSize.h – spacing)
```

If we know we are within the grid, we will then subtract the top-left spacing from mx and my, in preparation to convert the pixel mouse coordinates into coordinates within our grid, with a simple division by sw and sh:

```
      mx -= spacing;
      my -= spacing;
      x = mx / sw;
      y = my / sh;
```

One last check we'll add is to make sure we are not clicking on the grid lines themselves, as it would not be clear on which square we wish to position our pieces:

```
if((mx < sw   - lineWidth / 2 || mx > sw   + lineWidth / 2) && // 1st vertical line
   (mx < sw*2 - lineWidth / 2 || mx > sw*2 + lineWidth / 2) && // 2nd vertical line
   (my < sh   - lineWidth / 2 || my > sh   + lineWidth / 2) && // 1st horizontal line
   (my < sh*2 - lineWidth / 2 || my > sh*2 + lineWidth / 2))   // 2nd horizontal line
```

Then we are ready to place the X piece, if the square clicked by the user is empty, and request an update of our window:

```
         if(!board[y][x]) { board[y][x] = X; Update(null); }
```

Game Logic

A 2-players game is much more fun when there are 2 players. The eC distributed objects framework makes it extremely easy to write multi-player games without the tediousness of implementing a network protocol. Instead, a server connection class is defined and methods can be called across the network, as if the object was local. Many such samples can be found within the `samples/` directory of the SDK. For the purpose of this article however, we will focus instead on implementing an AI player. The human player will play X, while the computer plays O.

First, we will define a turn variable which specifies whose turn it is. A value of 0 will mean the game is over. We will initialize it to X: the player will start.

```
TTTSquare turn; turn = X;
```

Then to make sure the player can't continue playing after a TicTacToe, we will check whether it is indeed his turn to play (`turn == X`) within *`OnLeftButtonDown`*.

We will also turn our useless “TicTacToe” button into a “Reset” button that restarts the game, setting *`turn`* to X and clearing the *`board`* with 0s:

```
   Button btnReset
   {
      this, font = { "Arial", 12 }, caption = "Reset", position = { 8, 8 };

      bool NotifyClicked(Button button, int x, int y, Modifiers mods)
      {
         memset(board, 0, sizeof(board));
         turn = X;
         Update(null);
         return true;
      }
   };
```

Now we need code to detect a Tic Tac Toe!

```
   TTTSquare FindTicTacToe(TTTSquare state[3][3])
   {
      int i;

      // Diagonal '\'
      if(state[0][0] && state[0][0] == state[1][1] && state[1][1] == state[2][2])
         return state[0][0];
      // Diagonal '/'
      if(state[2][0] && state[2][0] == state[1][1] && state[1][1] == state[0][2])
         return state[2][0];

      for(i = 0; i < 3; i++)
      {
         // Horizontal
         if(state[i][0] && state[i][0] == state[i][1] && state[i][1] == state[i][2])
            return state[i][0];
         // Vertical
         if(state[0][i] && state[0][i] == state[1][i] && state[1][i] == state[2][i])
            return state[0][i];
      }
      return 0;
   }
```

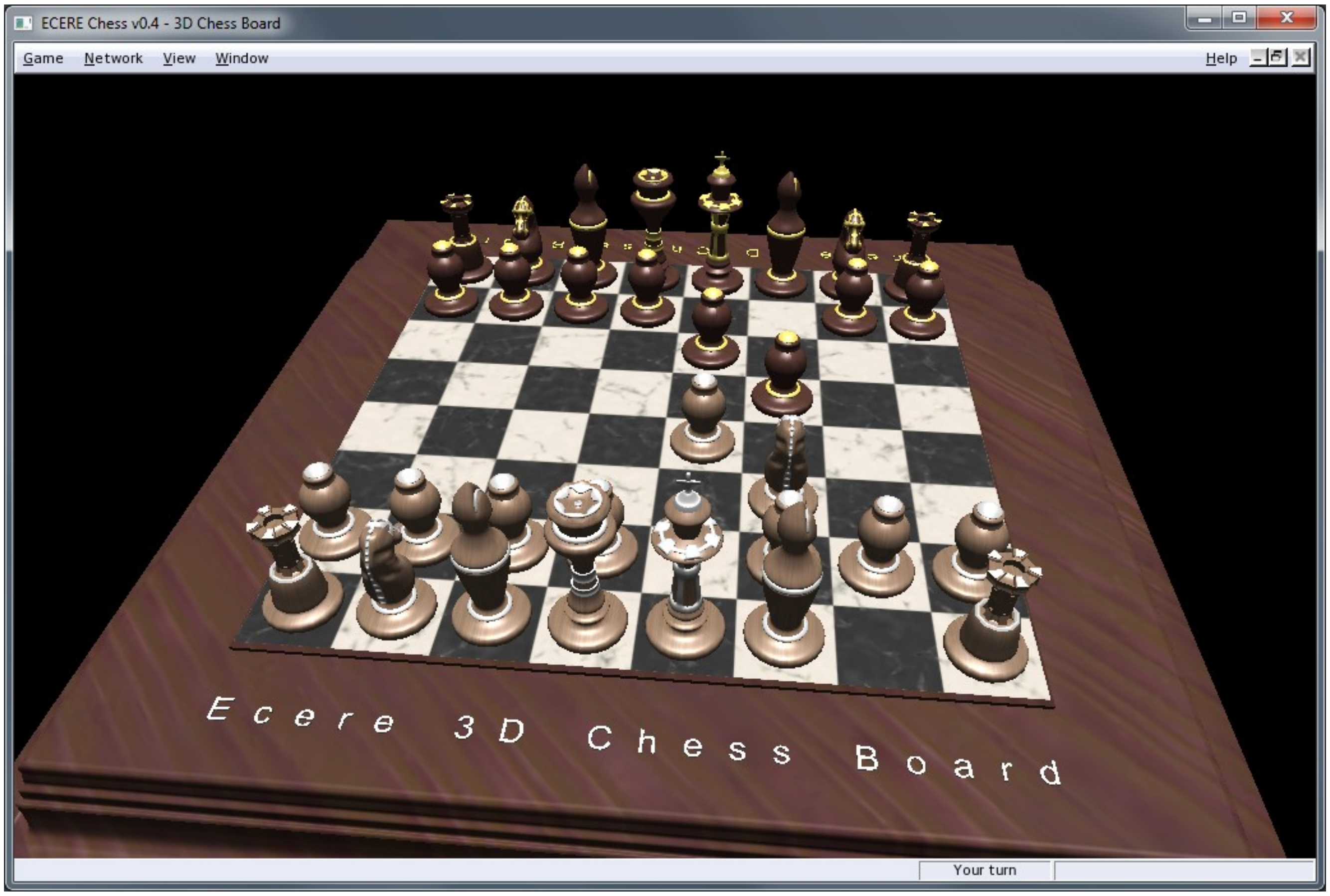
We will integrate the game logic within a *MovePlayed()* method which will get called right after the user places a piece on the board, in the *OnLeftButtonDown* method:

```
void MovePlayed()
{
   TTTSquare result = FindTicTacToe(board);
   if(result)
   {
      MessageBox { caption = "Tic Tac Toe!",
         contents = (result == X ? "You win!" : "Computer wins!") }.Modal();
      turn = 0;
   }
   else if(turn == X)
   {
      // Computer plays
      Point move { };
      turn = O;
      if(BestMove(turn, board, move) != noAvailableMove)
      {
         board[move.y][move.x] = O;
         MovePlayed();
      }
      else
         turn = 0;
   }
   else
      turn = X;
}
```

We check for a tic tac toe, if we found one, the game is over: we display the winner in a message box. If X just played, it is now the computer's turn to play. We call the *BestMove()* method where we will implement the computer's AI. If there was a move available, it gets played and *MovePlayed()* is invoked again to verify whether there is now a Tic Tac Toe. If the computer (O) just played, it is now the player's turn (X).

The AI

Implementing game AIs is always a fun endeavor! Classic 3x3 Tic Tac Toe is a rather simple game, and there are many approaches one could take to implement an AI, including hard-coding and/or categorizing solutions. However we will chose to implement a basic *minimax* algorithm, which can scale to more complex variants of the game, and can be used for other games as well.



The *Ecere Chess* application (whose source code is available on our *GitHub* page) also implements a *minimax* type algorithm. An overview of the minimax algorithm can be found at http://en.wikipedia.org/wiki/Minimax.

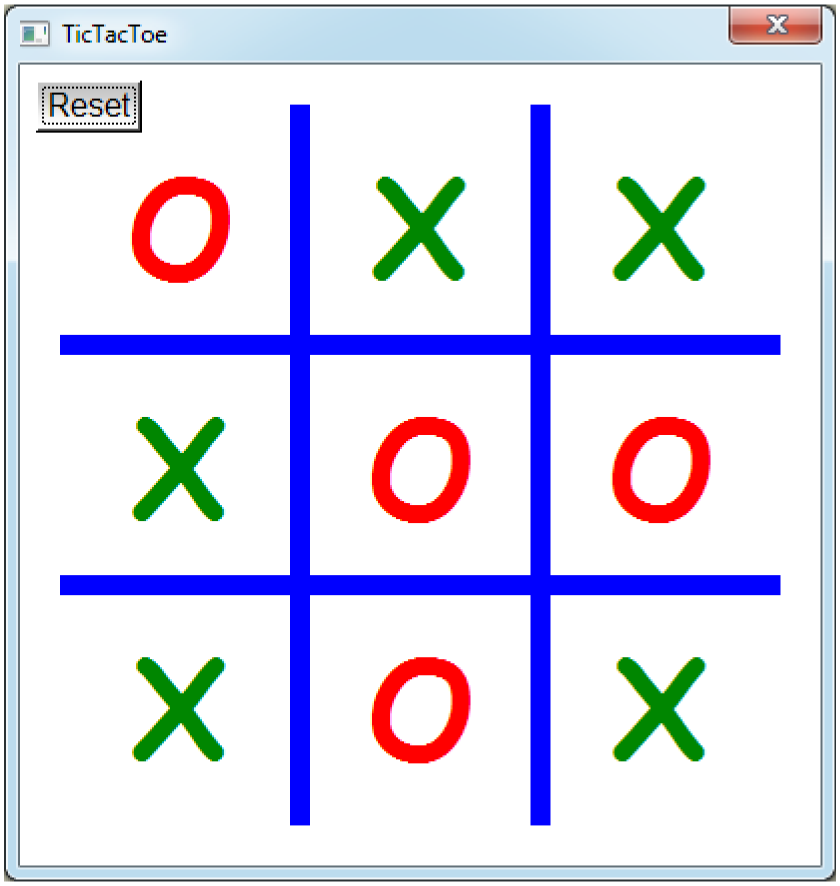
Here we will only provide a quick summary of the AI implementation. The AI included in the full listing at the end of this article includes additional code to add randomness and cause it to make human-like errors, based on a '*mastery*' level, ranging from 0 (dumb) to 100 (you can only tie). For the sake of understanding minimax, the simpler algorithm (which does not make mistakes) follows:

```
define noAvailableMove = -100;

   float BestMove(TTTSquare t, TTTSquare state[3][3], Point bestMove)
   {
      int x, y;
      float bestRating = noAvailableMove;
      for(y = 0; y < 3; y++)
      {
         for(x = 0; x < 3; x++)
         {
            if(!state[y][x])
            {
               float newRating;
               state[y][x] = t;
               if(FindTicTacToe(state))
                  newRating = 1;
               else
               {
                  Point move;
                  newRating = BestMove((t == X) ? O : X, state, move);
                  if(newRating == noAvailableMove)
                     newRating = 0;
                  newRating = -newRating/2;
               }

               state[y][x] = 0;
               if(newRating > bestRating)
               {
                  bestRating = newRating;
                  bestMove = { x, y };
               }
            }
         }
      }
      return bestRating;
   }
```

The code uses recursion to evaluate all possible moves, alternating between each player. It uses a floating point rating system, where the rating is negated at every player switch to make it relative to the current player. A TicTacToe at the current level is given a value of 1, while a TicTacToe further away is made less significant by a divide by 2. The best move is returned in the `bestMove` parameter. *No available move* is given a special value of -100.

Full Tic Tac Toe Listing

The full listing of `TicTacToe.ec` follows. With the Ecere SDK, it can be compiled and executed on any platform. A static binary on Windows (.exe) including the Ecere runtime library (with no external dependencies), takes up 657 KB once compressed with UPX.



```
import "ecere"

define spacing = 20;
define lineWidth = 10;
define mastery = 97;

define noAvailableMove = -100;

enum TTTSquare { _, X, O };

TTTSquare board[3][3];

class TicTacToe : Window
{
   caption = "TicTacToe";
   background = white;
   hasClose = true;
   clientSize = { 400, 400 };

   FontResource tttFont { "Comic Sans MS", 50, bold = true, window = this };
```

```
TTTSquare turn; turn = X;

TicTacToe()
{
   RandomSeed((uint)(GetTime() * 1000));
}

TTTSquare FindTicTacToe(TTTSquare state[3][3])
{
   int i;

   // Diagonal '\'
   if(state[0][0] && state[0][0] == state[1][1] && state[1][1] == state[2][2])
      return state[0][0];
   // Diagonal '/'
   if(state[2][0] && state[2][0] == state[1][1] && state[1][1] == state[0][2])
      return state[2][0];

   for(i = 0; i < 3; i++)
   {
      // Horizontal
      if(state[i][0] && state[i][0] == state[i][1] && state[i][1] == state[i][2])
         return state[i][0];
      // Vertical
      if(state[0][i] && state[0][i] == state[1][i] && state[1][i] == state[2][i])
         return state[0][i];
   }
   return 0;
}

float BestMove(TTTSquare t, TTTSquare state[3][3], Point bestMove)
{
   static int level = 0;
   int x, y;
   float bestRating = noAvailableMove;
   int filled = 0;
   bool couldTicTacToe = false;
/* A player is likely to see the opponent's tic tac toe in his own tic tac toe spot */
   Point badMove;
   Point moves[9];
   int numMoves = 0;

   level++;
   for(y = 0; y < 3; y++)
      for(x = 0; x < 3; x++)
         if(state[y][x]) filled++;

   for(y = 0; y < 3; y++)
   {
      for(x = 0; x < 3; x++)
      {
         if(!state[y][x])
         {
            float newRating;
            state[y][x] = t;
            if(FindTicTacToe(state))
               newRating = 1;
            else
```

```
               {
                  Point move;
                  newRating = BestMove((t == X) ? O : X, state, move);
                  if(newRating == noAvailableMove)
                     newRating = 0;
                  newRating = -newRating/2;
                  if(newRating <= -0.25f)
                  {
                     badMove = move;
                     couldTicTacToe = true;
                  }
               }

               state[y][x] = 0;
               if(newRating > bestRating)
               {
                  bestRating = newRating;
                  bestMove = { x, y };
                  numMoves = 1;
                  moves[0] = bestMove;
               }
               else if(level == 1 && newRating == bestRating)
                  moves[numMoves++] = { x, y };
            }
         }
      }
      if(GetRandom(0, 60) > mastery || (filled > 4 && filled < 7 && couldTicTacToe &&
            (bestMove.x != badMove.x || bestMove.y != badMove.y)))
      {
         if(level == 2 && GetRandom(0, 25) > mastery)
            bestRating = -0.5f;
         if(level == 4 && GetRandom(0, 100) > mastery)
            bestRating = -0.125f;
      }
      if(level == 1 && numMoves > 1)
         bestMove = moves[GetRandom(0, numMoves-1)];
      level--;
      return bestRating;
   }

   void MovePlayed()
   {
      TTTSquare result = FindTicTacToe(board);
      if(result)
      {
         MessageBox { caption = "Tic Tac Toe!",
                      contents = (result == X ? "You win!" : "Computer wins!") }.Modal();
         turn = 0;
      }
      else if(turn == X)
      {
         // Computer plays
         Point move { };
         turn = O;
         if(BestMove(turn, board, move) != noAvailableMove)
         {
            board[move.y][move.x] = O;
            MovePlayed();
      }
      else
         turn = 0;
   }
   else
      turn = X;
}

void DrawPieces(Surface surface)
{
   int sw = (clientSize.w - 2*spacing) / 3;
   int sh = (clientSize.h - 2*spacing) / 3;
   int x, y;
   int Xw, Xh, Ow, Oh;

   surface.font = tttFont.font;

   surface.TextExtent("X", 1, &Xw, &Xh);
   surface.TextExtent("O", 1, &Ow, &Oh);

   for(y = 0; y < 3; y++)
   {
      for(x = 0; x < 3; x++)
      {
         TTTSquare p = board[y][x];
         if(p == X)
         {
            surface.foreground = green;
            surface.WriteText(spacing + sw * x + sw / 2 - Xw/2,
                              spacing + sh * y + sh / 2 - Xh/2, "X", 1);
         }
         else if(p == O)
         {
            surface.foreground = red;
            surface.WriteText(spacing + sw * x + sw / 2 - Ow/2,
                              spacing + sh * y + sh / 2 - Oh/2, "O", 1);
         }
      }
   }
}

void OnRedraw(Surface surface)
{
   int sw = (clientSize.w - 2*spacing) / 3;
   int sh = (clientSize.h - 2*spacing) / 3;

   surface.background = blue;

   // Vertical lines
   surface.Area(spacing + sw   - lineWidth / 2,   spacing,
                spacing + sw   + lineWidth / 2-1, clientSize.h - spacing - 1);
   surface.Area(spacing + sw*2 - lineWidth / 2,   spacing,
                spacing + sw*2 + lineWidth / 2-1, clientSize.h - spacing - 1);
   // Horizontal lines
   surface.Area(spacing,                         spacing + sh   - lineWidth / 2,
                clientSize.w - spacing - 1, spacing + sh   + lineWidth / 2-1);
   surface.Area(spacing,                         spacing + sh*2 - lineWidth / 2,
                clientSize.w - spacing - 1, spacing + sh*2 + lineWidth / 2-1);
      DrawPieces(surface);
   }

   bool OnLeftButtonDown(int mx, int my, Modifiers mods)
   {
      if(turn == X && mx >= spacing && mx < clientSize.w - spacing && my >= spacing && my
< clientSize.h - spacing)
      {
         int sw = (clientSize.w - 2*spacing) / 3;
         int sh = (clientSize.h - 2*spacing) / 3;
         mx -= spacing;
         my -= spacing;
            /* 1st vertical line */
         if((mx < sw   - lineWidth / 2 || mx > sw   + lineWidth / 2) &&
            /* 2nd vertical line */
            (mx < sw*2 - lineWidth / 2 || mx > sw*2 + lineWidth / 2) &&
            /* 1st horizontal line */
            (my < sh   - lineWidth / 2 || my > sh   + lineWidth / 2) &&
            /* 2nd horizontal line */
            (my < sh*2 - lineWidth / 2 || my > sh*2 + lineWidth / 2))
         {
            int x = mx / sw;
            int y = my / sh;
            if(!board[y][x])
            {
               board[y][x] = X;
               Update(null);
               MovePlayed();
            }
         }
      }
      return true;
   }

   Button btnReset
   {
      this, font = { "Arial", 12 }, caption = "Reset", position = { 8, 8 };

      bool NotifyClicked(Button button, int x, int y, Modifiers mods)
      {
         memset(board, 0, sizeof(board));
         turn = X;
         Update(null);
         return true;
      }
   };
}

TicTacToe mainForm {};
```