Introduction to Game Programming

Session 2 – Drawing images and basic game structure

Nacho Cabanes
Nacho Iborra

IES San Vicente

Index of contents

1. Introduction

In this session we are going to see:

- How to draw images with Tao SDL. To do this, we are going to create our own *Image* class to deal with all the relevant information that we want to store for every image. Once we have this class, we will place some initial images in our video game to see how to change their positions and scale.
- How to get user input from the keyboard, detect the key that has been pressed and choose what to do.
- The basic structure of the video game. We are going to define three screens (welcome, game and credits), with some images on each, and we will let the user change among these screens by pressing some given keys.

You should have completed previous session (or downloaded the solution provided) in order to follow this session properly.

2. Drawing images

In this section we are going to learn how to deal with images in our video games. Remember that, as we said in Session #1, we need to have *SDL_image.dll* library added to our project.

2.1. Creating the Image class

First of all, let's create our own *Image* class in its corresponding source file *Image.cs*. It will have the following attributes:

- X and Y coordinates of the upper left corner where the image will be placed
- Image width and height
- An *IntPtr* object to store the image

We can define all these attributes as public properties, with the corresponding getter and setter:

```
class Image
{
	public short X { get; set; }
	public short Y { get; set; }
	public short ImageWidth { get; set; }
	public short ImageHeight { get; set; }
	public IntPtr ImagePtr { get; set; }
```

We define a constructor with the image file name, and the image width and height. With the file name, we will initialize the *IntPtr* object:

```
	public Image(string fileName, short width, short height)
	{
		ImagePtr = SdlImage.IMG_Load(fileName);
		if (ImagePtr == IntPtr.Zero)
		{
			Console.WriteLine("Image not found");
			Environment.Exit(1);
		}

		ImageWidth = width;
		ImageHeight = height;
	}
```

It will be useful to define a method called *MoveTo* that let us change the X and Y coordinates of the image whenever we want:

```
	public void MoveTo(short x, short y)
	{
		X = x;
```

```
            Y = y;
        }
```

2.2. Changes in Hardware class

To physically draw the image in the screen, we need to add some changes to our *Hardware* class (remember, this class is in charge of dealing with every hardware issue, such as communicating with the screen).

2.2.1. Draw the image

In this case, we are going to add a new method called *DrawImage* to this class. It will get an *Image* object as parameter, and it will draw this image in the specified X and Y coordinates, with the specified image width and height:

```
public void DrawImage(Image img)
{
      Sdl.SDL_Rect source = new Sdl.SDL_Rect(0, 0, img.ImageWidth,
          img.ImageHeight);
      Sdl.SDL_Rect target = new Sdl.SDL_Rect(img.X, img.Y, img.ImageWidth,
        img.ImageHeight);
      Sdl.SDL_BlitSurface(img.ImagePtr, ref source, screen, ref target);
}
```

Let's see these instructions more in depth:

1. First instruction defines a rectangle to determine which part of the image is going to be drawn. By default, we are going to draw the whole image, from its beginning (X = 0, Y = 0) to its end (width and height).
2. Second instruction sets the rectangle in the screen where the image is going to be drawn: from image X and Y coordinates to image width and height.
3. Last instruction copies the contents from the image (according to the *source* rectangle) to the screen, in the coordinates specified by the *target* rectangle.

2.2.2. View the image in the screen

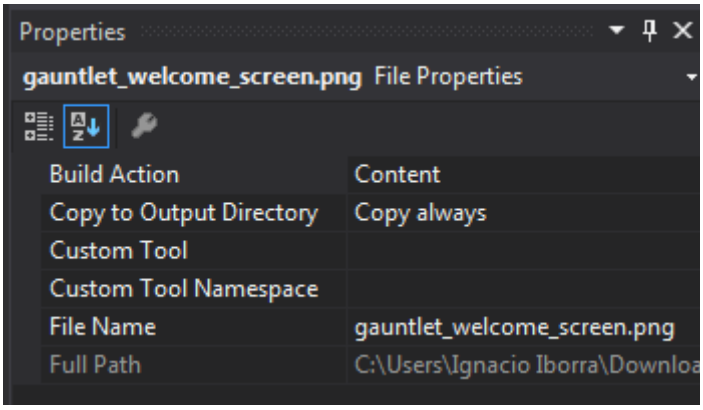
Besides, we must take into account that drawings are not directly performed in the screen itself, but in a secondary buffer. To update the screen with the things that we want to draw, we need to call a specific method.

To do this, we are going to add another method to our *Hardware* class. Let's call this method *UpdateScreen*. It will just call the *SDL_Flip* method from Tao SDL, responsible of updating the actual screen with the background drawings.

```
public void UpdateScreen()
{
      Sdl.SDL_Flip(screen);
}
```

2.3. Adding images to our project

Before testing these new classes and methods, we need to add the images to our project so that we can load them in the screen. We can do this following the same steps that we did to add the DLL files in previous session (copying and pasting them). Besides, you can create folders and subfolders from the *Project* > *New Folder* menu (if you have the project selected in the right panel). So you can create an *imgs* folder, for instance, and place there every image that we need. Remember to change the properties of each image and tell them to always copy to the output folder.



In this case, you will be provided with some images in the session resources. Copy them to *imgs* subfolder and update their properties.

2.4. Testing the changes

Let's see how this code works. Go to the *Main* method of our video game project and add the following lines to the existing code:

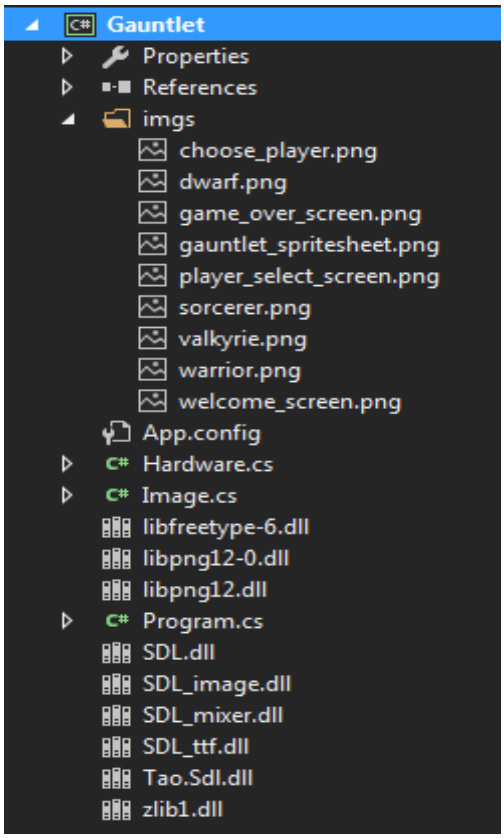
```
static void Main(string[] args)
{
      Hardware hardware = new Hardware(800, 600, 24, false);

      Image img1 = new Image("imgs/welcome_screen.png", 200, 200);
      Image img2 = new Image("imgs/warrior.png", 48, 48);

      img1.MoveTo(100, 100);
      img2.MoveTo(400, 200);

      hardware.DrawImage(img1);
      hardware.DrawImage(img2);
      hardware.UpdateScreen();

      Thread.Sleep(5000);
}
```

2.4.1. Adding additional DLLs

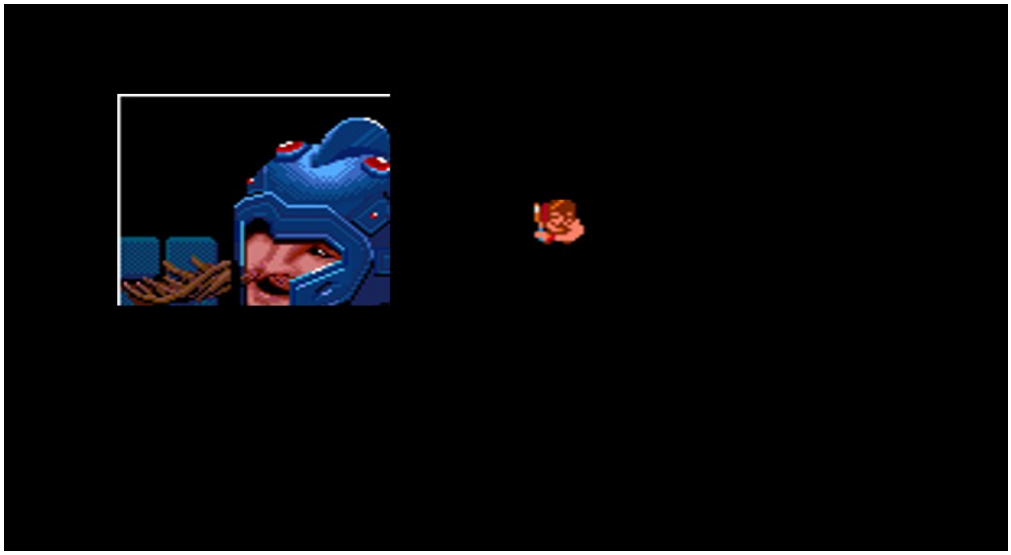
If you want to work with PNG files, you will need to use *libpng12.dll* and *libpng12-0.dll* libraries, that are already loaded in the project from previous session. But, if you want to deal with other image types, such as JPG images, you will need to add other library, called *jpeg.dll*. In both cases, you will also need *zlib1.dll* library, but this one is also added to our project from previous session.

2.4.2. Running the test

After making all these changes, your project folder should look like this:



If you run now the game, you will see something like this:



Notice how *warrior.png* image is drawn completely, whereas *welcome_screen.png* image is only partially drawn. This is because of the image width and height properties. In the case of *warrior.png* they are properly set (width 48, height 48), but with the welcome screen we have set a width (200) and height (200) that do not correspond with the actual image width and height, and that's why this image has been cropped.

2.5. Scaling images

After the test that we have run, you could think if there is a way to scale an image from the code, so that we make sure that it will always have the desired size. The answer with Tao SDL is no, and in general, it is not recommended to scale an image this way, because you can be wasting memory with too big images, and game performance may be affected due to the continuous scaling processes.

The best way of scaling an image is to use any image software, such as Gimp, and save it with the desired width and height.

3. Getting keyboard input

In this section we are going to learn how to get user input from the keyboard. To do this, we need to make some new changes to our *Hardware* class, and then we will test these changes in our project.

3.1. Changes in Hardware class

We need to define a method called *KeyPressed* in this class. It gets an event of KEYDOWN and returns the integer code of the pressed key.

```
public int KeyPressed ()
{
      int pressed = -1;

      Sdl.SDL_PumpEvents();
      Sdl.SDL_Event keyEvent;
      if (Sdl.SDL_PollEvent(out keyEvent) == 1)
      {
            if (keyEvent.type == Sdl.SDL_KEYDOWN)
            {
                  pressed = keyEvent.key.keysym.sym;
            }
      }

      return pressed;
}
```

What we do is basically detect an event with the first "if" sentence. If this event is a KEYDOWN (pressing a key down), then we return the code of the pressed key. Otherwise, this method returns -1.

We can also define some constants in this class, representing the keys that we want to manage in our video game. For instance:

```
public const int KEY_ESC = Sdl.SDLK_ESCAPE;
public const int KEY_UP = Sdl.SDLK_UP;
public const int KEY_DOWN = Sdl.SDLK_DOWN;
public const int KEY_LEFT = Sdl.SDLK_LEFT;
public const int KEY_RIGHT = Sdl.SDLK_RIGHT;
public const int KEY_SPACE = Sdl.SDLK_SPACE;
```

3.2. Testing the changes

Let's test this changes in our project. Replace the *Thread.Sleep* instruction at the end of the code with this *while* loop:

```
static void Main(string[] args)
```

```
{
      Hardware hardware = new Hardware(800, 600, 24, false);

      Image img1 = new Image("imgs/welcome_screen.png", 200, 200);
      Image img2 = new Image("imgs/warrior.png", 48, 48);

      img1.MoveTo(100, 100);
      img2.MoveTo(400, 200);

      hardware.DrawImage(img1);
      hardware.DrawImage(img2);
      hardware.UpdateScreen();

      while(hardware.KeyPressed() != Hardware.KEY_ESC);
}
```

If you run the game, you will see that you can only close it by pressing *Escape* key.

4. Game structure

Now, we are going to define some screens and classes that represent them, and a handler that controls which screen needs to be shown.

To be more precise, we are going to divide our video game in three screens:

- A welcome screen with the welcome screen image. When we press a given key (for instance, space bar), we will go to the game screen. If we press another given key (for instance, Escape), we will exit the game.
- The game screen, in which we will play Gauntlet
- A credits screen, in which we will show a credits or game over screen.

Besides, we need a handler that decides when to show each screen, depending on the game status and the user input.

4.1. The Screen class

As we are going to have several screen types in our video game, we are going to define a parent class for all of them. This class is going to be called *Screen*. It will have a *Hardware* object to manage the screen and draw things on it (depending on the screen type). It will also have a constructor to initialize the *Hardware* object and a virtual method called *Show* that will be overriden by its subclasses.

```
class Screen
{
	protected Hardware hardware;

	public Screen(Hardware hardware)
	{
		this.hardware = hardware;
	}

	public virtual void Show()
	{
	}
}
```

4.2. The WelcomeScreen class

Let's create a *WelcomeScreen* class in its corresponding source file (*WelcomeScreen.cs*). This class will inherit from previous class *Screen,* and will have a boolean attribute to indicate if user wants to exit the video game or not (apart from *hardware* attribute inherited from its parent class). Besides, it will store the welcome screen image to be drawn at this screen.

There will be a constructor to initialize the *Hardware* object (using *base* to call parent's constructor), set the *bool* attribute to false and load the image. Then, we will add an

overriden *Show* method (from parent class) to show this screen. Inside this method we must draw the image *welcome_screen.png* that will be provided to us, in the specified coordinates (0, 0). Then, we wait for the user to press a key, and decide what to do: go to game screen or exit the game. We can also define a *getter* to get the value of the boolean attribute

```
class WelcomeScreen : Screen
{
        bool exit;
        Image imgWelcome;

        public WelcomeScreen(Hardware hardware) : base(hardware)
        {
            exit = false;
            imgWelcome = new Image("imgs/welcome_screen.png", 800, 600);
            imgWelcome.MoveTo(0, 0);
        }

        public override void Show()
        {
            bool escPressed = false, spacePressed = false;
            hardware.DrawImage(imgWelcome);
            hardware.UpdateScreen();

            do
            {
                int keyPressed = hardware.KeyPressed();
                if (keyPressed == Hardware.KEY_ESC)
                {
                    escPressed = true;
                    exit = true;
                }
                else if (keyPressed == Hardware.KEY_SPACE)
                {
                    spacePressed = true;
                    exit = false;
                }
            }
            while (!escPressed && !spacePressed);
        }

        public bool GetExit()
        {
```

```
            return exit;
        }
}
```

Your welcome screen should look like this one when you test it from the main class:

4.3. The CreditsScreen class

Now, create the *CreditsScreen* class inside its source file *CreditsScreen.cs*. Its structure is very similar to the *WelcomeScreen* class, although it has no boolean attribute. It will have a constructor to initialize the inherited *hardware* attribute and the image to be shown, and an overriden *Show* method that draws the *game_over_screen.png* image in the screen, at the specified coordinates. Then, it will wait for the user to press spacebar key, before finishing.

```
class CreditsScreen : Screen
{
        Image imgCredits;

        public CreditsScreen(Hardware hardware): base(hardware)
        {
            imgCredits = new Image("imgs/game_over_screen.png", 800, 600);
            imgCredits.MoveTo(0, 0);
        }

        public override void Show()
        {
            hardware.DrawImage(imgCredits);
            hardware.UpdateScreen();
```

```
            while (hardware.KeyPressed() != Hardware.KEY_SPACE);
        }
}
```

The final appearance of this screen should be something like this (once you can test if from the main class):

4.4. The GameScreen class

Now, we are going to define the *GameScreen* class inside its source file *GameScreen.cs*. This class is also a subclass of *Screen.* We are going to put all the game loop and performance of the Centipede video game in this class, but this will be in later sessions. Now we are only going to define a constructor to initialize the inherited *hardware* attribute and an overriden *Show* method that draws the main character in the screen (image *character.png* that will be provided to you).

```
class GameScreen : Screen
{
        Image imgCharacter;
        public GameScreen(Hardware hardware): base(hardware)
        {
            imgCharacter = new Image("imgs/warrior.png", 48, 48);
            imgCharacter.MoveTo(380, 280);
        }

        public override void Show()
        {
            hardware.DrawImage(imgCharacter);
```

```
            hardware.UpdateScreen();

            while (hardware.KeyPressed() != Hardware.KEY_SPACE) ;
        }
}
```

Your game screen should look like this when you test it from the main class:

4.5. The GameController class

Now, we will define the *GameController* class in its source file *GameController.cs.* This class is NOT a subtype of *Screen* class. It will be in charge of deciding which screen needs to be shown at every moment during the game. It will have a *Start* method that will do the following, until we decide to exit the video game from the welcome screen

1. Show the welcome screen
2. If user does not want to exit:
    1. Show game screen
    2. Show credits screen

So this is the screen sequence that the game must have:

```
class GameController
{
      public void Start()
      {
            Hardware hardware = new Hardware(800, 600, 24, false);
            WelcomeScreen welcome = new WelcomeScreen(hardware);
            CreditsScreen credits = new CreditsScreen(hardware);
            GameScreen game = new GameScreen(hardware);
```

```
            do
            {
                  // Show welcome screen
                  // If user does not want to exit
                        // Show game screen
                        // Show credits screen
            } while(!welcome.GetExit());
      }
}
```

4.5.1. Clearing the screen

In the *Start* method of the *GameController* class, you may need to clear the screen before showing another one. To do this, you can add this method to the *Hardware* class...

```
public void ClearScreen()
{
      Sdl.SDL_Rect source = new Sdl.SDL_Rect(0, 0, screenWidth,
            screenHeight);
      Sdl.SDL_FillRect(screen, ref source, 0);
}
```

... and call it before showing any screen in the *Start* method. For instance:

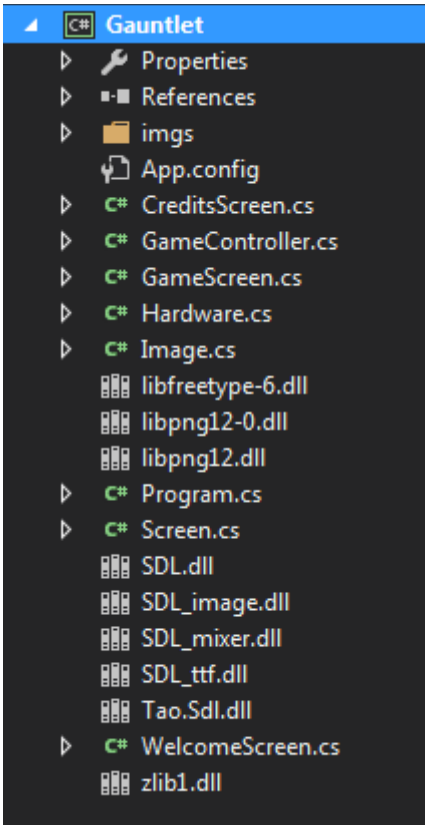
```
hardware.ClearScreen();
welcome.Show();
...
```

4.6. The main class

With all these classes already created, our main class (*Centipede* class) will only have to create a *GameController* object and start it

```
class Centipede
{
      static void Main(string[] args)
      {
            GameController controller = new GameController();
            controller.Start();
      }
}
```

At the end of all these steps, your project should look like this:


Gauntlet
Properties
References
imgs
App.config
CreditsScreen.cs
GameController.cs
GameScreen.cs
Hardware.cs
Image.cs
libfreetype-6.dll
libpng12-0.dll
libpng12.dll
Program.cs
Screen.cs
SDL.dll
SDL_image.dll
SDL_mixer.dll
SDL_ttf.dll
Tao.Sdl.dll
WelcomeScreen.cs
zlib1.dll

5. To think a little...

To finish with this session, you must add an additional screen to the game screen flow. After the welcome screen, you should show a **PlayerSelectScreen** class that shows these images (*player_select_screen.png* and *choose_player.png* to point at the chosen player):

Inside this screen, you must choose a player with the up and down arrow keys (*choose_player.png* image must move to the chosen player every time you press a key), and, when pressing the space bar, you must move to GameScreen and draw the image of the chosen player (either *warrior.png*, *valkyrie.png*, *sorcerer.png* or *dwarf.png*). For instance, if we choose the dwarf in PlayerSelectScreen, then our GameScreen should look like this one: