Learn C++ with Esenthel

Mute (http://mutecode.com)

Revision: November 12, 2016

Contents

Part I: 2D Concepts

Learning to program is much more fun when you see some results on the screen. Esenthel can be used for 2D as well as 3D applications, but 2D is the easiest to start with. By the end of this part, you will be able to draw and manipulate basic shapes, images and text. And just for fun, you will learn to add some sound.

Chapter 1

Positions in 2D

Positions in 2D have an x- and a y- coordinate, just as a point in a graph. In Esenthel the zero point is the middle of the screen. Positive values on the X-axis are to the right, negative values to the left. Positive values on the Y-axis are in the upper half of the screen, negative values on the lower half.

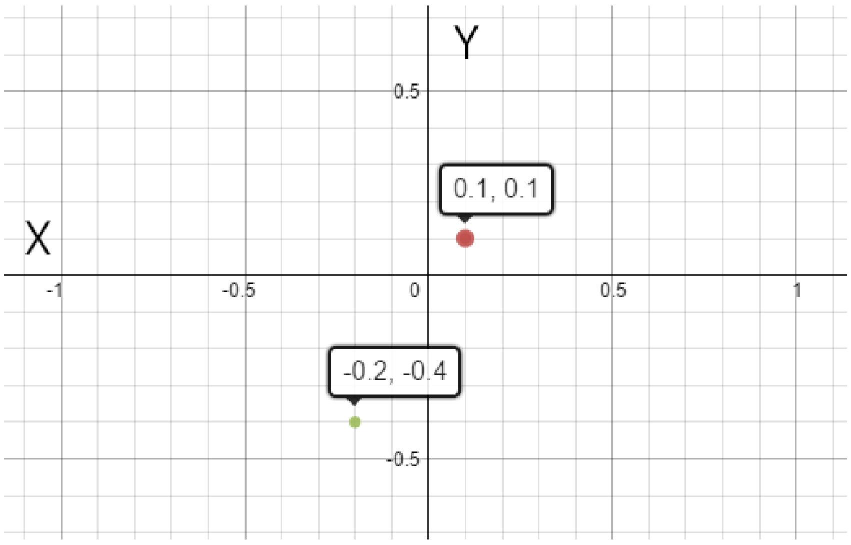

Figure 1.1: 2D coordinates.

The class `Vec2` is used to represent a 2D coordinate. There are several ways to set the x and y values:

```
Vec2 pos;
pos.x =  0.1;        // set the x value to  0.1
pos.y = -0.3;        // set the y value to -0.3
```

```
pos    =  0.5;        // both x and y are set to 0.5
pos.set(0.1, -0.3); // use the function set(float x, float y)
                    // to assign both x and y
```

Time for Action

Where would these positions be located on the screen?

1.1 Show a position on the screen

Although the class `Vec2` is mainly used to calculate positions, it is also possible to show it on the screen. This can be done with the function `draw(Color)`. The argument should contain the color in which the coordinate must be drawn.

```
Vec2 p1(0.2, 0.4); // Create a point p1, assigning x and y
                   // with the constructor.
Vec2 p2;           // Create a point p2.

void InitPre()
{
   EE_INIT();
}

bool Init()
{
   p2.set(-0.2, -0.5); // assign values to p2 with the set function
   return true;
}

void Shut() {}

bool Update()
{
   if(Kb.bp(KB_ESC)) return false;

   return true;
}

void Draw()
{
   D.clear(BLACK); // Clear the screen
   p1.draw(RED  ); // draw p1 in red
   p2.draw(BLUE ); // draw p2 in blue
   Vec2(0 ,0).draw(GREEN); // create a temporary object and draw
                           // it in green
}
```

Time for Action

Write this code in the editor. Don't copy/paste it: you won't learn anything from copying code. Make sure it runs without error. If not, compare your code with this example. Learning to understand errors is very important. Try to understand what went wrong and why.

1.2 Math

You can do math with `Vec2`, just as you can with plain numbers. It is possible to assign x and y values directly, but you can also do math with the class itself. In this case the operator will be applied to x as well as y:

```
Vec2 p1(0.1,  0.3);
Vec2 p2(0.3, -0.1);
Vec2 p3 = p1 + p2;  // x: 0.4 , y: 0.2
p3 -= 0.1;          // x: 0.3 , y: 0.1
p3 *= 2;            // x: 0.6 , y: 0.2
p3 = p1 / 2.f;      // x: 0.05, y: 0.15
```

1.3 Points to remember

The position `Vec2(0,0)` always stands for the middle of the screen. But the borders are not always clear. Not every computer screen has the same size, but you probably want to know the border values are. For example, you want to draw something in the right corner of the screen. For this reason, Esenthel provides a few functions in the object `D` (display). You can request the width of the screen with `D.w()` and the height of the screen with the function `D.h()`. This makes it very easy to calculate the next points:

```
Vec2 middle(0,0);
Vec2 left(-D.w(), 0);
Vec2 right(D.w(), 0);
Vec2 rightUpperCorner(D.w(), D.h());
Vec2 leftUpperCorner(-D.w(), D.h());
```

If you'd like to draw a point at a distance of 0.1 from the left upper corner, you could do that like this:

```
Vec2(-D.w() + 0.1, D.h() - 0.1).draw(PINK);
```

1.4 Summary

- ❏ Points in 2D have an x- and y-coordinate. In Esenthel you will use the class `Vec2` to store a point.
- ❏ The class `Vec2` has a function `draw(Color)` to draw a point on the screen.
- ❏ You can do math with `Vec2`, just as with an ordinary number.
- ❏ `Vec2(0,0)` will always be the middle of the screen.
- ❏ The borders of the screen can be calculated with `D.w()` and `D.h()`.

Time for Action

Create an application which draws these points:

1. A white point in the middle of the screen.
2. A red point on 0.1 units away from the left border of the screen.
3. A blue point on 0.2 units from the upper right corner.
4. A yellow dot on 0.35 units from the lower border of the screen, on 2/3 of the total screen width.

Chapter 2

Interaction

Most interactions in an application are triggered by keyboard and mouse. (Mobile devices mostly use touches, and we'll discuss those later.) In this chapter we'll talk about how to handle them.

2.1 Mouse Interaction

You can find the class `Mouse` in Esenthel Engine ⇒ Input ⇒ Mouse. Operating systems cannot handle more than one mouse pointer, which is why Esenthel already has a mouse object defined: `Ms`. You do not need to declare an instance of `Mouse`.

When you look at the functions provided in this class, you'll see a lot of them. Most of the time you will be interested in clicks and positions.

2.1.1 Position

The position of the mouse pointer on the screen is a `Vec2`. The function `Ms.pos()` returns the current position. The following code example shows the current mouse position on the screen.

```
Vec2 mousePos;

void InitPre()
{
   EE_INIT();
}
```

```
bool Init()
{
   return true;
}

void Shut() {}

bool Update()
{
   if(Kb.bp(KB_ESC)) return false;
   mousePos = Ms.pos();
   return true;
}

void Draw()
{
   D.clear(BLACK);
   mousePos.draw(RED);
}
```

TIME FOR ACTION

Use the example above, but instead of showing this position right below the mouse pointer, show it a little bit below.

2.1.2 Clicks

You can use the `Ms` object to access anything related to the mouse. Most computer mouses have more than one button, which is why you need to indicate which button you'd like to query. The left button has an index of 0, the right button an index of 1. If the mouse wheel can also be pressed, it will have index 2. Here are some examples:

```
Ms.bp(0); // true if the left button was pressed in this frame
Ms.br(1); // true if the right button was released in this frame
Ms.b (0); // true as long as the left button is held down
Ms.bd(0); // true when a double click is detected in this frame
```

There is a subtle difference between `Ms.bp()` and `Ms.b()`:

- `Ms.bp()` Is only true on the moment the button is pressed. The button might still be held down in the next frame, but the function will return false by then.
- `Ms.b()` This function will return true as long as you don't release the button.

The next example will illustrate this difference:

```
Vec2 mouse1Pos;
Vec2 mouse2Pos;

void InitPre()
{
   EE_INIT();
}

bool Init()
{
   return true;
}

void Shut() {}

bool Update()
{
   if(Kb.bp(KB_ESC)) return false;

   if(Ms.bp(0)) {
      mouse1Pos = Ms.pos();
   }

   if(Ms.b(1)) {
      mouse2Pos = Ms.pos();
   }

   return true;
}

void Draw()
{
   D.clear(BLACK);
   mouse1Pos.draw(RED  );
   mouse2Pos.draw(GREEN);
}
```

The red dot will only update its position on the moment you press the left button. In contrast, the green point will update as long as the right button is pressed. While working on an application, you will often have to decide which approach is best for the action at hand.

Time for Action

Starting with this example, make it so that the second mouse position is shown only when the right mouse button is being held down.

2.1.3 Mouse Wheel

Most recent mouse have a wheel. The wheel can be turned, but it doesn't have an absolute position. Esenthel can only tell you how much the wheel was turned since the last update. This distance is called a 'delta'. In general, a delta value indicates the difference between the previous and the current value. There are delta values for time, position changes, mouse wheel changes and so on. For the mouse wheel you will need the function `Ms.wheel()`. The result is a float value.

The next example shows a dot on the vertical axis, controlled by the mouse wheel. The function `Ms.wheel()` only returns the change since the previous update, which means it cannot be assigned to the position directly. You will have to add it to the current value. This change is a very small positive number while you move the wheel upwards. While moving down, the value will be negative.

```
Vec2 mousePos;

void InitPre()
{
   EE_INIT();
}

bool Init()
{
   return true;
}

void Shut() {}

bool Update()
{
   if(Kb.bp(KB_ESC)) return false;
   mousePos.y += Ms.wheel() * 0.1;
   return true;
}

void Draw()
{
   D.clear(BLACK);
   mousePos.draw(RED);
}
```

Time for Action

Start from the example above. When the left mouse button is held down, you need to control the horizontal dot position. When the right mouse button is held down, you control the vertical position of the dot.

2.1.4 Cursor

The mouse cursor is actually just an image shown at the screen. It is possible to change this image in your application:

```
Ms.cursor(Images( ===drop cursor image here=== ));
```

Time for Action

Modify the cursor image in the previous exercise. There are few suitable images in the folder gfx ⇒ mouse. Show the image 'BGNormal' at the application start. When a mouse button is held down, show the image 'BGMove' instead.

2.1.5 Other functions

The `Mouse` contains a lot of functions we didn't talk about. Just for fun, try to figure out what these functions do. The first and second one are rather obvious, but you might want to experiment a bit to understand what the last one does.

- ❑ `Ms.hide()`
- ❑ `Ms.show()`
- ❑ `Ms.eat()`

2.2 Keyboard Interactions

The computer keyboard is often used to interact with a game. Later on, in the chapter about the GUI, you will see how to use a keyboard to type in text. This chapter will concentrate on using keystrokes to control interaction with your application.

Every key has a name. Once you know the name, you are able to check on that particular key. Careful though: all names assume a qwerty keyboard. If your keyboard has another layout, like azerty, you will want to keep an image of a qwerty layout at hand. This seems a bit weird, but when controlling a game, the actual position is is more importance than the character shown on the keyboard. For instance: we often use the WASD keys to control movement. Now imagine this would translate to the same characters on an azerty keyboard. That wouldn't be practical at all!

That said, checking on a keystroke isn't really much different from a mouse click. You'll see that most methods are quite similar.

2.2.1 Key Methods

Take your time to examine these methods:

```
Kb.bp(KB_N) // true if N was pressed down in this frame.
Kb.br(KB_E) // true if E was released in this frame.
Kb.b (KB_R) // true while R is being held down
Kb.bd(KB_D) // true if a double tap occurred on D
```

Even though we use the object `Kb` instead of `Ms`, most functions are exactly the same. But instead of using the number of a mouse button as an argument, you will now enter a keyboard code. This code will always start with KB_, followed by a character or a number. Other options are:

Functions	Control	Modifiers	Arrows	Numpad
KB_F1	KB_ESC	KB_LCTRL	KB_LEFT	KB_NPDIV
KB_F2	KB_ENTER	KB_RCTRL	KB_RIGHT	KB_NPENTER
...	KB_SPACE	KB_LSHIFT	KB_UP	KB_NP1
KB_F12	KB_BACK	KB_RSHIFT	KB_DOWN	KB_NP2
	KB_TAB	...		...

A complete list of all keys can be found at Esenthel Engine ⇒ Input ⇒ Input Buttons.

Time for Action

Create an application with a dot on the screen. Use the arrow keys to move this dot. Every time an arrow key is pressed, you move this dot 0.1 units in that direction.
The dot itself will be drawn in green, unless the spacebar is held down. If so, draw the dot in red.

2.2.2 Gradual Changes

The method `Kb.bp()` can be used for sudden changes. For example when you open a window, add an object, show a menu or close the application. Like this:

```
if(Kb.bp(KB_F1)) CreateObject();
```

Imagine you would use the method `Kb.b()` instead. As long as the F1 key is held down, a new object will be created. On a fast computer, this would be about 60 times a second. And even when you're really fast and release the key almost instantly, you will probably create more than one object.

Still, this is a usable method, but only when you need gradual changes. Like when you move a dot to the right as long as a key is held down:

```
if(Kb.b(KB_RIGHT)) point.x += 0.01;
```

Every frame, the value of x will increase a little bit. The dot will gradually move to the right.

Time for Action

Adapt your last exercise to use the `Kb.b()` method. The dot will now move very fast, so you might want to decrease the added value.

2.2.3 Delta Time

The exercise above has one major problem: the position will be updated at every frame. Should you test this application on two different computer, one with a good graphics card and 60FPS, and another without a dedicated graphics card running at 30FPS, the movement will be twice as fast on the good computer! We won't mind for a small exercise, but this could be a serious gaming disadvantage for the girl with the slower computer.

We can fix this problem with the concept of **delta time**. The delta time is the time that has passed between this update and the previous one. *(Remember the mouse wheel delta, being the distance traveled by the mouse wheel between this frame and the previous one.)* The delta time can be retrieved with the next method:

```
Time.d();
```

Would you like to move an object at a speed of 1 unit per second? Try this code:

```
if(Kb.b(KB_RIGHT)) point.x += 1 * Time.d();
```

Time for Action

Adapt the previous exercise, so that the delta time is used. Once you've tested your app, replace the number one with a float variable defined on top of your application. Use two extra keys to alter the variable and hence control the movement speed.

CHAPTER 3

Text

There's no doubt you will need to show text on a screen eventually. Esenthel provides the class `Str` to remember characters, words and even entire paragraphs.

There are several ways to put text into a `Str`:

```
Str text("hello world"); // use the constructor
text  = "hello "; // use assignment
text += "world" ; // use the addition assignment operator
```

NOTE

The text between quotes is officially called a 'string literal'. When you don't have to do anything with a string but drawing it on screen, you can often just use a string literal.

3.1 Drawing Text

Now let's try and draw some text on the screen. Contrary to the class `Vec2`, the `Str` class does not have a 'draw' method. The object `D` is used to show text on the screen:

```
Str myString;
myString = "hello world";
D.text(Vec2(0, 0), myString); // place text in the middle of the screen
D.text(Vec2(0, -0.1), "Een string literal"); // or use a string literal
```

The second argument of the method `text` is the text itself. The first argument is the desired position. By now, you know enough about coordinates to understand why this has to be a `Vec2`.

Time for Action

Create an application that shows the words ‘left’, ‘right’, ‘top’ and ‘bottom’ on a logical screen position. Hide the mouse pointer and instead show the word ‘mouse’ at the current mouse position.

3.2 Combining Text and Numbers

You cannot just combine text and numbers in C++. As far as your computer knows, the values 42 and “42” have nothing to do with each other. The first one is an integer, the second one a string literal which happens to contain the characters 4 and 2.

Now, the `Str` class in Esenthel helps you a lot. It allows you to assign and add numbers to strings. For instance, all these expressions are correct:

```
int i = 42;
Str myString =  42;
myString    +=   i;
myString    +=  42;
myString    += 4.2;
```

But when you combine string literals (text between quotes) with numbers, you’re in trouble. The compiler will try to combine them first, before dumping them in a string. And it can’t do that. So this won’t work:

```
Str myString;
myString = "score: " + 42;
myString = 42 + "score: ";
```

Without going into details here, there is a simple solution: a `Str` object called `S` is provided by Esenthel. Whenever you run into an error when you try to combine text and integers or floats, start with `S +`.

```
Str myString;
myString = S + 42  + " is your score";

int myScore = 10;
D.text(Vec2(0, 0), S + "score: " + myScore);

// more complex
D.text(Vec2(0, 0), S + "score: " + myScore + " / 10");
```

Time for Action

Create an application with an int 'score'. Increase the score every time you press the space bar. Show the current score on the screen.

3.3 Vec2 as Text

While debugging your application it can be useful to show the x- and y-coordinates of a `Vec2` on the screen. You could do this the hard way:

```
Vec2 pos = Ms.pos();
D.text(Vec2(0, 0), S + "x: " + pos.x + " y: " + pos.y);
```

However, you're certainly not the first programmer to draw a `Vec2` on screen to check on its value. That's why the class as a method to make your life a little easier:

```
Vec2 pos = Ms.pos();
D.text(Vec2(0, 0), S + pos.asText());
// or without the extra step:
D.text(Vec2(0, 0), Ms.pos().asText());
```

Time for Action

Create an application with a dot which can be moved by means of the arrow keys. Show the position of the dot on the screen.

3.4 Font and Size

Drawing text is all well and good, but the default text probably isn't gonna do much for your game design. To customize this, you will need to provide your desired font. A new font can be added by right-clicking in the file-tree and choosing 'New Font'. A dialog will open to change the font's settings.

Next you create a new textStyle. With a textStyle, you can define a color and alignment for a particular font. *(In the 'font' drop down menu, the active font can be set.)* Even with one font, you can create as many textStyles as you like.

Now if you'd like to use your newly created textStyle to display text in your application, you can use an alternate version of `D.text()` :

```
D.text(*TextStyles(=== drop style here ===), Vec2(0, 0), "text");
```

Just drag your textStyle over to the code editor and drop it as an argument of `TextStyles()`. Remember there's an asterisk right before this function. We'll discuss it later, but for now just remember it has to be there.

Time for Action

Experiment with different options to shape a font and text style. Draw at least four different texts on the screen, each one with their own style.

CHAPTER 4

Shapes

Once you know how to show a dot on the screen, other shapes are easy. (If you don't, review chapter 1.) Most mathematical shapes can be drawn on the screen. The classes to do that are in the green folder Esenthel Engine ⇒ Math ⇒ Shapes.

Not all available shapes are intended for display in 2D. Some classes, like `Ball` and `Tube`, are intended for 3D development. For now, we will focus on 2D shapes like `Circle`, `Edge` (line), `Quad`, `Rectangle` and `Triangle`.

4.1 Circle

To draw a circle you need a radius(r) and a position(pos). The radius is a `float`, the position a `Vec2`. There is more than one way to pass these to a circle:

```
// using the constructor, with r(float), pos.x(float), pos.y(float)
Circle c(0.1, 0, 0);

// using the constructor, with r(float), pos(Vec2)
Circle c(0.1, Vec2(0, 0));

// during the coarse of the application
c.set(0.1, 0, 0);
c.set(0.1, Vec2(0, 0));

// directly changing the variables
c.r = 0.1;
c.pos = Vec2(0, 0);
```

4.1.1 Methods

The set function aside, there are also methods available to retrieve the current area or perimeter, and to draw the circle on the screen.

```
// retrieve area and perimeter
float a = c.area();
float b = c.perimeter();

// draw a blue circle on the screen
c.draw(BLUE);

// draw the perimeter only
c.draw(BLUE, false);
```

There's something remarkable about the `draw` method! It can be used with one as well as with two methods. To see how this is possible, look at the declaration of this method, which can be found at Esenthel Engine ⇒ Math ⇒ Shapes ⇒ Circle:

```
void draw(C Color & color, Bool fill = true, Int resolution = -1) C;
```

The first argument is a `Color`. The second argument is a `bool` named 'fill'. You might suspect this means whether or not you desire to draw the circle as a perimeter or an area. And of course you're right. Only, the argument doesn't stop there: it has a value assigned (= true). This is called a default value. If you agree with the default, you don't have to explicitly pass true as an argument. Only when you don't agree, you will have to pass false.

Note

The third argument (resolution) is also optional. Experiment with several values to find out what is does. (Try values like 2, 3, 8, 14, ...)

4.1.2 Math

You can do math with circles. There are operators like `+=`, `-=`, `/=` and `*=`, which can be used to alter the radius or the position. How do you know which value will be altered? Well, if you add a float to a circle, the radius will change. When you add a `Vec2` this will alter the position.

```
Vec2 pos(0.1, 0);
Circle c(0.1, pos);

// move the circle 0.1 units to the right
c += pos;
// move the circle 0.2 units down
```

```
c -= Vec2(0, 0.2);
// double the radius
c *= 2;
```

Time for Action

Recreate the following image by drawing circles on the screen. If you like a challenge, try creating a more fitting mouth with the method `drawPie`.

4.2 Edge2

An `Edge2` is a line. Just as with `Vec2`, the '2' is important. The code will still be valid when you use an `Edge` instead of an `Edge2`, but that class is intended for drawing in 3D instead. To define an `Edge2`, you need two points (`Vec2`), being the beginning and the end. Esenthel offers two methods to pass these to the object:

```
Edge2 e1;
Edge2 e2;

// using newly created vectors ...
e1.set(Vec2(0, 0), Vec2(0.1, 0.1));

// ... or existing vectors
Vec2 pos1(0.3, 0.6);
Vec2 pos2(0.1, 0.2);
e1.set(pos1, pos2);

// set all x and y values as floats
e2.set(-0.4, 0.2, -0.7, 0.9;
```

4.2.1 Methods

An `Edge2` provides several neat methods. The most obvious will be `draw()`:

```
line1.draw(PURPLE    ); // draw a purple line
line2.draw(GREEN, 0.1); // draw a green line, with width 0.1
line3.draw(GREEN, RED); // draw a line that starts out green but fades to
    read
```

Other methods of interest are `center()`, `delta()`, `dir()` and `length()`. All of them can be found in ⇒ Math ⇒ Shapes ⇒ Edge.

4.2.2 Exercises

Time for Action

The methods `Sin()` en `Cos()` allow you to retrieve the sine and cosine from any value. This is most fun when that value is the current time. The result over time is a value which moves smoothly between -1 and 1.

```
float x = Sin(Time.curTime());
float y = Cos(Time.curTime());
```

1. The code above illustrates how to use the sine and cosine with the current time. Create an application which shows a line, starting from the middle of the screen towards the current value of sine and cosine for the x and y value of the ending.
2. Change the width of the line.
3. Make the line move at double speed.
4. Decrease the length of the line.
5. *(This will be a bit harder)* Try to create an analog watch.

Time for Action

Take a look at Esenthel Engine ⇒ Math ⇒ Shapes ⇒ Edge. Examine the method `lerp(float s)`. This function needs an argument between zero and one, and return a position on the line.

1. Define an `Edge2` and a `float` with value zero.
2. In the init function, assign a start and end position to your line.
3. In the update function you increase the float with `Time.d()`. When the float value is larger than 1, is should be assigned 0.
4. Draw the line in black with a width of 0.05. Construct a `Vec2` with the result of `lerp()`. The argument of the method should be your float. Draw this point in red, also with a width of 0.05.

Time for Action

Draw a triangle on the screen.

4.3 Rect

The last shape in the chapter is a rectangle: `Rect`. You will end up using this shape quite a lot, because a rectangle happens to be the shape of most gui elements, like buttons, windows and images.

After the last exercise, you know that you need 3 positions to draw a triangle. Logic dictates a rectangle requires 4 positions, right? But there's one property of a rectangle that makes it a lot easier: all corners have an angle of 90%. This means that when you pass the lower left corner and the upper right corner, there is enough information to find the two remaining corner positions.

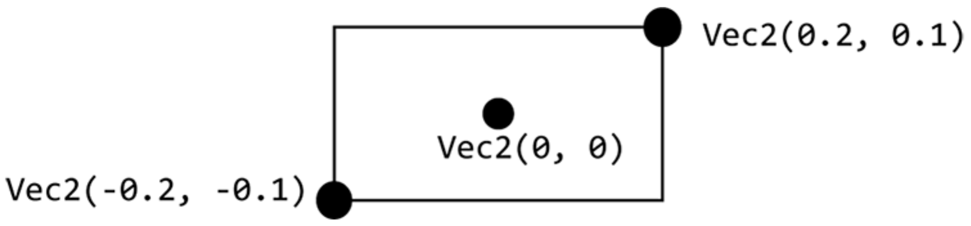

Figure 4.2: The corners of a rectangle

This means a rectangle can be created like this:

```
Rect(Vec2(-0.2, -0.1), Vec2(0.2, 0.1)).draw(BLACK, false);
```

Again there is a draw function with the color as the first argument. And optionally you can draw only the perimeter of the rectangle. Just as with `Edge2` there is another version of the constructor allowing you to enter the x- and y-coordinates one by one.

```
Rect(-0.2, -0.1, 0.2, 0.1).draw(BLACK);
```

Time for Action

Recreate this screen in an application:

4.3.1 A moving Rectangle

If you want to draw an image on the screen, it will require a rectangle. As such, a rectangle will be the basis of almost every element on your screen. It will often come in handy to have a special class for movable rectangles. As an example, we'll create a rectangle which can be moved with the arrow keys.

Rectangles have one big disadvantage when you try to move them. Instead of a central position, both corners have to be moved. This is why we often use a Vec2 to remember the center position. As an extra, this class will remember its color.

```
class movableRect {
    Vec2 pos;
  Color color;
}
```

This class can be extended with a create, update and draw methods. The create method can be used to pass the color and the starting position to the object. This starting position already has a default value, so you can use the create method without it.

```
void create(C Color & color, C Vec2 & pos = 0) {
    T.color = color;
  T.pos = pos;
}
```

Note

There some other new stuff in this method. The & sign indicates that the color and pos will be passed as a reference. The character C means we will not change the value that was passed into the method. You will learn more about this in one of the next chapters. Within the function, the character T is used to solve a practical problem. The class and the method both have a variable named 'color'. This makes it impossible for the compiler to know what variable you mean. You could rename one of them, but that makes the code harder to read. A more elegant solution is to add T. before the class variable. T stands for 'this' and by that we mean 'this class'. The version without T. will be the local variable, passed into the method.

Within the update method, the position can be changed by using the arrow-keys. You came across this code before, in chapter 2.2.2.

```
void update() {
  if(Kb.b(KB_LEFT )) pos.x -= Time.d();
  if(Kb.b(KB_RIGHT)) pos.x += Time.d();
  if(Kb.b(KB_UP   )) pos.y += Time.d();
  if(Kb.b(KB_DOWN )) pos.y -= Time.d();
}
```

The final thing we need is a draw function. Notice that this is the only place we actually create a rectangle. That's because we don't really need the whole rectangle in the other methods. This could be different if, for example, we need to check if the rectangle hits another object. So you could add a real Rect to this class, but right now it is not needed.

```
void draw() {
  Rect(pos - Vec2(0.2, 0.1), pos + Vec2(0.2, 0.1)).draw(color);
}
```

As you see, we use the position 'pos' when we create the rectangle. The position will be the middle of the rectangle. But we still need to define the corner. This can be done be subtracting and adding the same value from the position. In this case the value is literal, but a more advanced class could just define a `Vec2 size` and use that instead. That would make it easy to define movable rectangles of different sizes.

The whole class is listed again below. It is the first time in this course we actually create a new class, so be sure you understand what is going on. And pay attention to the keywords `private` and `public`. You should know those from an introductory c++ course. We will talk more about those later on, but remember they are part of a good programming style.

```
class movableRect {
private:
  Vec2 pos, size;
  Color color;

public:
  void create(C Color & color, C Vec2 & pos = 0, C Vec2 & size = Vec2(0.1,
     0.1)) {
    T.color = color;
    T.pos   = pos  ;
    T.size  = size ;
  }

  void update() {
    if(Kb.b(KB_LEFT )) pos.x -= Time.d();
    if(Kb.b(KB_RIGHT)) pos.x += Time.d();
    if(Kb.b(KB_UP   )) pos.y += Time.d();
    if(Kb.b(KB_DOWN )) pos.y -= Time.d();
  }

  void draw() {
    Rect(pos - size, pos + size).draw(color);
  }
}
```

Time for Action

Add a pink square to the application, using the newly created class `movableRect`. *(A bit harder)* Be sure the square cannot move outside of the screen.

4.4 Cuts

A function that is used very often in combination with shapes is `Cuts`. There are quite a lot of different versions of this function, but its meaning is always the same: do two shapes overlap, or don't they? The `Cuts` function allows you to check if a dot is inside a circle, if a circle (partially) overlaps a rectangle, if a line cuts a rectangle and so on.

To check if a dot is inside a circle, use the code below:

```
Vec2 pos(0.1, 0.1);
Circle area(0.3, Vec2(0));

if(Cuts(pos, area)) area.draw(RED);
```

Of course, in this example it is already certain the dot will always be inside the circle. But you could use the same principle to see if the mouse pointer is inside the circle, like this:

```
Circle area(0.1, Vec2(0));

if(Cuts(Ms.pos(), area)) area.draw(RED);
else area.draw(BLACK);
```

The code above is the starting point for many possible interactions. Often, this will be in combination with other rules. Try to find out what the code below does:

```
Rect button(Vec2(-0.2, -0.1), Vec2(0.2, 0.1));
bool hover = false;

// in the update function:
if (Cuts(Ms.pos(), button)) {
    hover = true;
  if(Ms.bp(0)) exit();
} else hover = false;

// in the draw function:
if(hover) {
  button.draw(Color(0, 255, 0));
} else {
  button.draw(Color(0, 155, 0));
}
```

Time for Action

1. Create an application with 3 circles, each one below the other. Only the border of the circles is drawn, unless the mouse pointer is in that particular circle. In which case you draw a filled circle.
2. Make the circles move slowly to the right as long as the mouse pointer is inside the circle.
3. Create an integer 'score'. When the circle goes over the right side of the screen, increase the score by one and place the circle back in the middle.

Time for Action

(A bit harder) Create your own class that behaves like a button, with a hover effect. The button also needs a line of text, which will be set by the create function.

Chapter 5

Images and Sound

5.1 Images

A modern 2D game will almost always contain images. Whatever happens on the screen, it mostly comes down to showing and manipulating images. And because every image is a rectangle, you will use the `Rect` class to show them on the screen.

```
Images(=== drop image here ===).draw(Rect(-0.1, -0.1, 0.1, 0.1));
```

You can use `Images()` to refer to any image in your project. The image in question can be dropped as the function argument, between the parentheses. Once that is done, you use the function `draw` with a `Rect` argument to show the image on the screen. It is very easy to make your image move this way. The only thing your application has to remember is the current position. The actual rectangle can be derived from that point.

```
Vec2 ship(0, -0.8);

// during update:
if(Kb.b(KB_LEFT )) ship.x -= Time.d();
if(Kb.b(KB_RIGHT)) ship.x += Time.d();

// during draw:
Images(=== spaceship ===).draw(Rect(ship - 0.1,  ship + 0.1));
```

Note

When you're looking for images like the one in this example, just use google images. Mostly you will want images with a transparent background. This will be an image in GIF or PNG format. But Esenthel doesn't support GIF, so PNG is your target of choice. The search tools on Google Images allow you to search specifically for transparent images. Can't find what you're looking for? Try adding the term 'icon' or 'sprites' to your query. Just remember this is great while you're experimenting. But once you're working on a real game you should not use images which aren't yours, unless you really verified their license permits the use you intend.

To add realism to your game it is a good idea to use variations on an image. In the next example, two alternate versions of the same image are used during movement.

```
if(Kb.b(KB_LEFT))
{
  Images(=== spaceship ===      ).draw(Rect(ship - 0.1,  ship + 0.1));
} else if(Kb.b(KB_RIGHT))
{
  Images(=== spaceship_left === ).draw(Rect(ship - 0.1,  ship + 0.1));
} else
{
  Images(=== spaceship_right ===).draw(Rect(ship - 0.1,  ship + 0.1));
}
```

Time for Action

Find 3 very sad images and create an application with a moving image. Make sure the images are switched one way or another.

Another way to add some dimension is by varying the image over time. You actually create a little animation by rotating through a list of images all the time. The following example presents a player class with three variations for every direction.

```
class player
{
private:
   Vec2 pos;
   float timer = 0.4;
   DIR_ENUM dir = DIRE_DOWN;

public:
   void update()
   {
      // adjust direction
      if(Kb.bp(KB_UP   )) dir = DIRE_UP   ;
      if(Kb.bp(KB_DOWN )) dir = DIRE_DOWN ;
      if(Kb.bp(KB_LEFT )) dir = DIRE_LEFT ;
      if(Kb.bp(KB_RIGHT)) dir = DIRE_RIGHT;
      // update position
      switch(dir)
      {
         case DIRE_UP   : pos.y += Time.d() * 0.5; break;
         case DIRE_DOWN : pos.y -= Time.d() * 0.5; break;
         case DIRE_LEFT : pos.x -= Time.d() * 0.5; break;
         case DIRE_RIGHT: pos.x += Time.d() * 0.5; break;
      }

      // animation timer
      timer -= Time.d();
      if(timer < 0) timer = 0.4;
   }

   void draw()
   {
      // pointer to an image
      ImagePtr current;

      // evaluate direction
      switch(dir)
      {
         case DIRE_UP:
         {
            // pick an image according to time (changes between 1 - 2 - 3
               - 2)
            if      (timer > 0.3) current = Images(=== back1 ===);
            else if (timer > 0.2) current = Images(=== back2 ===);
            else if (timer > 0.1) current = Images(=== back3 ===);
            else                  current = Images(=== back2 ===);
            break;
         }

         case DIRE_DOWN:
         {
            if      (timer > 0.3) current = Images(=== front1 ===);
            else if (timer > 0.2) current = Images(=== front2 ===);
            else if (timer > 0.1) current = Images(=== front3 ===);
            else                  current = Images(=== front2 ===);
            break;
         }

         case DIRE_LEFT:
         {
            if      (timer > 0.3) current = Images(=== left1 ===);
            else if (timer > 0.2) current = Images(=== left2 ===);
            else if (timer > 0.1) current = Images(=== left3 ===);
            else                  current = Images(=== left2 ===);
            break;
         }

         case DIRE_RIGHT:
         {
            if       (timer > 0.3) current = Images(=== right1 ===);
            else if (timer > 0.2) current = Images(=== right2 ===);
            else if (timer > 0.1) current = Images(=== right3 ===);
            else                  current = Images(=== right2 ===);
            break;
         }
      }

      // show the current image on the screen
      current->draw(Rect(pos - 0.05, pos + 0.05));
   }
}
```

Time for Action

The example above can be found in the course template. Create an object of the player class and use it in an application to see the result of this code.

Note

On line 39 there's an object 'current' of the class `ImagePtr`. this class is a 'pointer' to an `Image`. The code below that line will make the pointer 'point' to the image we want to show next. At the bottom, `current->draw()` is used to draw that image. Note the arrow instead of the dot. This is a sign that current is not a real image object, but just points to one. (Don't worry if you find this hard to grasp. You will learn more about pointers in another chapter.)

Time for Action

The `Image` class in Esenthel contains a whole bunch of functions. Most of them you will not need any time soon, but it is good to remember that whatever you want to do with your image, there's a good chance there is a function which has you covered.
For now, experiment a bit with the functions below to learn about their intent.

- ❏ `draw` has a version which allows you to pass some colors as an argument. Try this out. (Hint: the second color will very often be `TRANSPARENT`)
- ❏ Draw an image using `drawFit`. How does this differ from `draw`?
- ❏ Draw an image using `drawRotate`. Try rotating the image with the arrow keys.
- ❏ Draw an image using `drawFS`
- ❏ (a bit harder) Load an image, apply a blur and export as PNG. Can you do it?

5.2 Sound

To make your game a bit more attractive you will want to add sound. Generally speaking, there are two groups: music and effects (FX). Music will mostly be played in the background while FX is linked to certain actions like pressing a button or dying horribly.

5.2.1 Music

A soundtrack can be played with the class `Sound`:

```
Sound soundtrack;

void InitPre()
{
   EE_INIT();
}

bool Init()
{
   soundtrack.create(=== drop your audio file here ===);
   return true;
}

void Shut() {}

bool Update()
{
   if(Kb.bp(KB_ESC)) return false;

   if(Kb.bp(KB_SPACE))
   {
      if(soundtrack.playing())
      {
         soundtrack.pause();
      } else
      {
         soundtrack.play();
      }
   }
   return true;
}
```

There's a few things to remember, though:

- ❏ Before you can use a sound, it must be loaded from disk. This is done with the `create()` function. It needs at least one argument: the audio file. Like with images, you can simply drag the file from your resources on to your function. You will want to do this inside of the `Init()` function, because you don't want to load your file from disk at every update.

- `play()` will cause the sound to start playing.
- `pause()` will pause the sound. Who would have guessed, right? When you use play after pause, the sound will continue right where it left off.
- Instead of `pause()` you can also use `stop()`. Now when you start playing again, the sound will start from the beginning.

The `create()` function also has a few optional arguments. Here's an example with all of them:

```
soundtrack.create(=== audio file ===, true, 0.8, VOLUME_MUSIC);
```

But what do they mean, little grasshopper?

1. The first argument is known. That's the audio file.
2. the second argument is the loop value. It can be true or false and is used to indicate if you'd like the sound to 'loop'. (Which mean it will start from the top when it is finished.) The default value is false.
3. Next comes the volume. Volume scales from zero to one, with a default of 1.
4. The last argument is a 'channel'. Esenthel has several channels for playing audio. If you don't use this argument, the sound will use the channel 'VOLUME_FX'. It is generally a good idea to use several channels for different types of sounds, because the volume of a channel can be changed. It makes it easy to implement volume changes for music, fx or voices.

Time for Action

Create an application which loads a soundtrack. Draw a green, an orange and a red circle on the screen. The track should start playing when you click the green circle, pause when you click the orange circle and stop when you click the red one.

Time for Action

Extra: Search the header file of the sound class for a method to retrieve the current playing position within a sound file. Draw this position on the screen.

Time for Action

Extra: This will be a bit harder. Use the functions `fadeInFromSilence()` and `fadeOut()` to apply a fade of 3 seconds instead of an immediate start and stop.

5.2.2 Playlists (Extra)

To play music, you can also use playlists. This will bring more variation to your soundtrack, and also allows you to switch between playlists when the mood of the game changes. Just examine the code below to see how it works:

```
// defined play lists
Playlist Battle , // battle playlist
         Explore, // exploring playlist
         Calm   ; // calm playlist

void InitPre()
{
   EE_INIT();
}

bool Init()
{
   if(!Battle.songs()) // create 'Battle' playlist if not yet created
   {
      Battle += (=== drop action music ===); // add "battle0"
      Battle += (=== same here ===); // add "battle1"
   }
   if(!Explore.songs()) // create 'Explore' playlist if not yet created
   {
      Explore+= (=== drop tranquil music ===); // add "explore" track
   }
   if(!Calm.songs())  // create 'Calm' playlist if not yet created
   {
      Calm   += (=== a very relaxed soundtrack ===); // add "calm"
   }
   return true;
}

void Shut()
{
}

bool Update()
{
   if(Kb.bp(KB_ESC))return false;
   if(Kb.c('1'))Music.play(Battle );
   if(Kb.c('2'))Music.play(Explore);
   if(Kb.c('3'))Music.play(Calm   );
   if(Kb.c('4'))Music.play(null   );
   return true;
}

void Draw()
{
   D.clear(TURQ);
```

```
    if(Music.playlist()) // if any playlist playing
    {
       D.text(0, 0, S+"time " +Music.time()+" / "+Music.length()+" length");
    }else
    {
       D.text(0, 0, "No playlist playing");
    }
    D.text(0, -0.2, "Press 1-battle, 2-explore, 3-calm, 4-none");
}
```

5.2.3 FX

Short sounds can also be played with the method **SoundPlay()**. This will play the sound directly, without requiring you to create an object of the class **Sound**. Because there is no object you won't have any control over the sound after you started it. Therefore this technique will mostly be used for very short effects, such as footsteps.

Time for Action

In the template project you will find a few sounds in the folder 'sound'. Create an application that plays back a 'blip' every time you push the arrow-down key. Use the 'rotate' sound for arrow-left and arrow-right. And last, play back the 'down' sample when you press the space bar.
Extra: Add the soundtrack again, but this time control the volume of the track with the mouse wheel.

Part II: Basics

In this part you will learn about more generic programming concepts like random values and containers. You will also review some elementary stuff like references, enumerations, functions and classes and see some typical Esenthel use of those. To finish the part off, a chapter about application states is added. Mainly because it didn't really fit in anywhere else.

Chapter 6

Random

A game will rarely stay interesting when it is completely predictable. To avoid predictability, developers use random values provided by the **Random** class.

6.1 Whole Numbers

The function **Random()** returns a random number between 0 and 4.294.967.295. Check this yourself with the next example:

```
uint number = 0;

bool Update() {
  if(Kb.bp(KB_SPACE)) number = Random();
  return true;
}

void Draw() {
  D.clear(BLACK);
  D.text(0, 0, S + number);
}
```

You will rarely need a number this big. Which is why you can use the **Random()** function with one or more arguments. When used with one argument, the function will return a number in the range 0 to the argument minus one. In other words, **Random(5)** will return one of the values 0, 1, 2, 3 or 4. Count them, that's 5 different values. A common beginners mistake is to expect the number five as a result. That will never, ever happen!

It is also possible to pass two arguments. `Random(-2, 4)` returns one of these values: -2, -1, 0, 1, 2, 3 or 4. The important thing to remember that with this version, the arguments are inclusive.

Time for Action

Create the basics of a lottery application. Show a new number from 1 to 42 (inclusive) on the screen every time you press the space bar.

Note

How to get a random color? The RGB values which make up a color have a value between 0 and 255. To randomize a color, try this:

```
Color myColor;
myColor.set(Random(256), Random(256), Random(256));
```

6.2 Random Float

So far we've discussed random whole numbers. But you will often need floating point values. These are provided by the function `RandomF()`. This function will return a value between 0 and 1 by default, but it will also accept arguments. `RandomF(3)` will return a value between 0 and 3. `RandomF(-1.3, 2.5)` returns one between -1.3 and 2.5.

These functions are often used to show an object on a random position. Like so:

```
Circle c;

c.pos.x = RandomF(-D.w(), D.w());
c.pos.y = RandomF(-D.h(), D.h());
```

In the example above, we won't even pass actual numbers to the function `RandomF()`. Instead, we let our application decide. The width and height of the screen are not the same on every device. By asking the engine about it, we will always end up with the correct values.

Time for Action

Create an application which shows a circle on the screen. Every time the space bar is pressed, you change the circle's position.

Time for Action

Create an application with a small rectangle on the screen. Assign a new position every second.

Time for Action

Starting from the previous exercise, add an int ‘score’ equal to zero. Draw this score somewhere on the screen. When the left mouse button is pressed, check if the mouse pointer is on top of the rectangle. If it is, increase the score by one.

Time for Action

(A bit harder) With every score increase, the position of the rectangle should change a bit faster.

Chapter 7

Containers

So far, you needed to define all global objects at the top of your application file. This is OK for a little exercise, but when your project grows in size, this becomes a problem. You also have to know upfront how many objects you need. Even for a little game like asteroids, it is impossible to know how many rocks there will be on the screen at all times.

When you need several objects of the same type, you can use a container. When you declare a container for a certain object type, you can add objects to this container during the course of the application. An easy to use container is `Memc`. The declaration of a container requires that you provide the type of objects it will contain. When you need a container for floats, you would declare it as a `Memc<float>`. A container for rectangles would be a `Memc<Rect>`. Look at this code for an example of a container with circles:

```
// Declare a container for circles
Memc<Circle> circles;

void InitPre()
{
   EE_INIT();
}

bool Init()
{
     // add 10 circles to this container
  for(int i = 0; i < 10; i++)
    {
      // The method New() adds a new circle to the container.
      // At the same time the Circle method set() is used to
      // assign a radius and a position.
        circles.New().set(0.1, RandomF(-D.w(), D.w())
                             , RandomF(-D.h(), D.h())
        );
    }
    return true;
}

void Shut() {}

bool Update()
{
    if(Kb.bp(KB_ESC)) return false;
    return true;
}

void Draw()
{
    D.clear(BLACK);

    // Go over all circles in the container and
    // draw them on the screen.
    for(int i = 0; i < circles.elms(); i++)
    {
        circles[i].draw(RED);
    }
}
```

Time for Action

What would happen if, by mistake, you place the code to generate circles in Update instead of Init?

Time for Action

Put this code back in Init, but add code to the Update function: every time you press the space bar, an extra circle should be added to the container.

Time for Action

Show an image on the screen instead of a circle. *(Too hard? Start with a rectangle!)*

7.1 New()

The method `New()` creates a new element at the end of the container. At the same time, it returns a reference to this new element, which is why can use the `set()` method of circle in the example above.

But suppose you need to use two methods of the newly created object? You could try something like this:

```
for(int i = 0; i < 10; i++)
{
  circles.New().set(0.1, RandomF(-D.w(), D.w()), RandomF(-D.h(), D.h()));
  circles.New().extend(-0.05);
}
```

...but it won't work. Instead you are creating two new circles at every iteration. The method `set` is called on the first circle, the method `extend` at the second. The solution is simple: Pass the result of `New()` to a temporary variable. The type of this variable must be a reference to a circle.

```
for(int i = 0; i < 10; i++)
{
  Circle & c = circles.New();
  c.set(0.1, RandomF(-D.w(), D.w()), RandomF(-D.h(), D.h()));
  c.extend(-0.05);
}
```

Note

If you don't know what a reference is, don't worry. We'll talk about it later. For now, remember that you need to put an ampersand (&) between the type and the name.

7.2 Using objects

Very often, you need to iterate over all elements in a container. For example when you draw them all on the screen. It would be very annoying if you had to remember somehow exactly how many elements a container contains. Fortunately, you do not have to. Containers provide a method `elms()` which returns the current number of elements. And to access individual elements you can use square brackets, just like with primitive C arrays.

```
for(int i = 0; i < circles.elms(); i++)
{
  circles[i].draw(RED);
}
```

Because you will need an iteration like this very, very often, Esenthel provides a 'shortcut'. A macro `REPA` exists to replace the whole for-loop declaration with one instruction:

```
REPA(circles)
{
  circles[i].draw(RED);
}
```

Remember this as ‘repeat all’. *(Or don’t remember it at all. Plain for-loops will always work just as well.)* And you can do more with this than just draw every element on the screen. Take a look at the next example and try to figure out what it does.

```
REPA(circles)
{
  circles[i].pos.y += Time.d();
  if(circles[i].pos.y > D.h()) {
    circles[i].pos.y -= (2*D.h() + RandomF(1));
  }
}
```

Time for Action

- ❏ Test the code above in an application. What function would you place this code in?
- ❏ Add a function to add an extra circle every time you hit the space bar.
- ❏ Instead of a fixed radius, use a random value between 0.01 and 0.1.
- ❏ Draw only the perimeter of the circle, in white, on a blue background.
- ❏ If there are any people nearby, shout out loud what this looks like.

7.3 Adding Objects

You will add objects to a container quite a lot. This might happen in the Init function as well as the Update function. Next are a few examples to get you started, but there are a lot of different ways to add objects. It is up to you to figure out what is the best approach in your application.

7.3.1 During Init

Ten circles on random positions:

```
for(int i = 0; i < 10; i++)
{
  Circle & c = circles.New();
  c.set(0.1, RandomF(-D.w(), D.w()), RandomF(-D.h(), D.h()));
}
```

Circles from the left to the right side of the screen:

```
for(float i = -D.w(); i < D.w(); i += 0.2) {
  circles.New().set(0.1, i, 0);
}
```

Squares placed evenly over the screen:

```
for(float i = -D.w(); i < D.w(); i += 0.2)
{
  for(float j = -D.h();  j < D.h();  j += 0.2)
  {
     rects.New().set(i - 0.05, j - 0.05, i + 0.05, j + 0.05);
  }
}
```

Time for Action

Test all of the examples above and make sure you understand every one of them. Always add code to display all elements on the screen.

7.3.2 During Update

Respond to keyboard input:

```
if(Kb.bp(KB_SPACE)) {
  circles.New().set(  RandomF(0.05, 0.2)
                    , RandomF(-D.w(), D.w())
                    , RandomF(-D.h(), D.h())
  );
}
```

Use the mouse position:

```
if(Ms.bp(0)) {
  circles.New().set(0.05, Ms.pos());
}
```

With a timer:

```
Flt timer = 3; // Put this line on top of the file.

// These lines belong in Update()
if(timer > 0) timer -= Time.d();
else {
  timer = 3;
  circles.New().set(  RandomF(0.05, 0.2)
                    , RandomF(-D.w(), D.w())
                    , RandomF(-D.h(), D.h())
  );
}
```

Time for Action

Test all of the examples above and make sure you understand every one of them. Always add code to display all elements on the screen.

7.4 Removing Objects

Of course you also want to remove objects from a container. Which is not that hard:

```
Memc<Vec2> dots;

// ... add a lot of dots

dots.remove(0); // remove the first dot
```

With the method `remove` and the index of the element as an argument, you delete an object in a container. Be careful though. Very often you will want to remove an element while iterating over a container. It is a common beginner mistake to alter an object after you've deleted it:

```
for(int i = 0; i < dots.elms(); i++) {
  if(dots[i].y < -D.h()) {
    dots.remove(i);
  }
  dots[i].y -= Time.d();
}
```

In the example above, all dots are moved down at every update. When a dot arrives at the bottom of the screen, it will be removed from the container. After removing a dot, it is not the current dot that is moved down, but the next one in the container. This is not a big problem, unless this was actually the last dot in the container. In which case you try to move down an object past the end of the container. The result will be a program crash, your computer might explode and probably a kitten will die somewhere.

To prevent this from happening it is a good rule to put the remove method as the last statement in the loop:

```
for(int i = 0; i < dots.elms(); i++) {
  dots[i].y -= Time.d();
  if(dots[i].y < -D.h()) dots.remove(i);
}
```

Things start to be a bit more complicated when you combine more than one container. In the next example we have container for dots and a container for circles. The code tries to verify if a dot hits a circle. If this is the case, both the circle and the dot must be removed from their container. To do this, we have to check every dot against every circle.

```
for(int i = 0; i < dots.elms(); i++) {
  for(int j = 0; j < circles.elms(); j++) {
      if(Cuts(dots[i], circles[j])) {
        dots.remove(i);
        circles.remove(j);
        // At this point there is one less dot in the container, but
        // the remaining circles will still be compared against the
           current dot.
        // current dot. If we are at the last dot, it will no longer be
           valid.
        // To prevent a crash, we add a break statement to go back to
        // the outer for loop:
        break;
    }
  }
}
```

And if you'd like to clear all container elements at once:

```
dots.clear();
```

7.5 A little Game

1. Create a triangle at the bottom of the screen. This triangle can be moved back and forth with the arrow keys.
2. Add a container for the class `Vec2`. Every time you press the space bar, you add an element on the location of the triangle.
3. Increase the y value of every container element in the Update function.(Use `Time.d()`!) If an element reaches the top of the screen, remove it from the container.
4. Draw all elements on the screen in the `Draw()` function.
5. Create a second container for circles. Every second a circle must be added somewhere at the top of the screen.

6. Move all circles down in the `Update()` function.
7. Show all circles in the `Draw()` function.
8. When a circle hits a `Vec2` from the other container, both must be removed.
9. When a circle hits the triangle, ‘Game Over’ must be shown on the screen.

You could go even further with this game. Don’t create new circles after the game is finished, and disable movement and shooting. Circles might move faster the longer you play, a score can be shown or you might give the player more than one life.

And instead of triangles and circles, images might be used. Have fun!

CHAPTER 8

Classes: the basics

The basic idea behind classes is to group variables which belong together. For example, `Vec2` has the values x and y which together make up a 2D position. Without this class, you would have to declare two separate floats to remember a position.

A software library is, for the most part, a collection of classes that work well together. The library developer provides classes which can be used to make your job easier. It would be hard to imagine a game without positions, so it makes sense that Esenthel engine provides you with a class to store such a position. In a most basic form, this class could look like this:

```
class Vec2 {
  float x;
  float y;
}
```

When this class is defined, it can be used anywhere in your application:

```
Vec2 pos;
Vec2 pos2 = pos; // assigns the values of pos to pos2
pos.x = 3;       // change the x value of pos
pos.y = pos2.x;  // assign the x value of pos2 to the
         // y value of pos
```

In every application except from the small exercises we have done in the previous chapters, you should design your own classes. Almost every aspect of your program should be contained within a class. So it's about time you learn more about them.

NOTE

When two or more objects or variables belong together, they should be part of a class.

Please note that there's a difference between a class and an object. In the example above, `Vec2` is a class, but pos and pos2 are objects of this class. You cannot use `Vec2` as if it were an object, but you can use this class to create one:

```
Vec2.x = 3; // wrong!
Vec2 pos;
pos.x = 3; // correct.
```

8.1 Create your own class

Imagine you need a moving circle in your application. To have a circle move at a fixed speed, you will need a variable to store this speed. It clearly belongs with the circle, but it isn't part of the Circle class provided by Esenthel because not every circle needs to move. So we create our own class:

```
class movingCircle {
  Circle c;
  Vec2 speed;
}

// declare an object of this class
movingCirle mc;

// some code in Init()
mc.c.set(0.1, Vec2(-0.4, -0.3));
mc.speed = Vec2(0.3, 0.5);

// some code in Update()
mc.c.pos += mc.speed * Time.d();
```

Once you have a class `movingCirle`, you can create as much moving circles as you like, all able to remember their own speed. *(Although the current class can be improved a lot, which you will learn in a next chapter).*

Time for Action

1. Create a container for the class `movingCircle`. Every time you press the space bar an element should be added to the container, on a random position and with a random speed.
2. Make sure all circles update their position in the Update function. When a circle goes outside of the screen, put it back in the middle.
3. Expand the class `movingCirle` with a color. Every object has a random color, to be used to draw it on the screen.

Note

More work will be done on this exercise in the next chapter. Save it!

CHAPTER 9

Methods

Methods, or class functions, can be added to a class. Mostly, methods should be the way to interact with your class. Instead of changing member variables directly, a class should provide methods to access and alter them. Appart from providing accessors to variables, methods should also be used for any kind of action your class might perform.

9.1 Methods without arguments or result

A plain method does exactly the same thing, every time you call it. When you think back the the class `movingCircle`, you might want to add a method to put the circle back in the middle of the screen:

```
class movingCircle {
  Circle c;
  Vec2 speed;

  void reset() {
    c.pos = Vec2(0);
  }
}

// somewhere in your application
movingCircle mc;
mc.reset();
```

In this example, the circle's position will be set to zero when the `reset()` method is called. This method consists of three parts:

void Indicates the method has no result. (More about results in a moment.)

reset The name of the method. You can pick your own name, but of course the restrictions for variable names also apply here. (Some characters are not allowed, don't start with a number, ...)

() End with a couple of braces. Later on, arguments will be put between them.

Time for Action

Add the `reset()` method above to the previous exercise. In the `Update()` function of your program, replace the code to reset the circle's position with your method.
Add a method `draw()` to the same class. This method will draw the circle with a certain color. Again, replace the code in your app with this method.

9.2 Methods with Arguments

A method can have one or more arguments. Arguments are used to pass values to a method. In the previous example, the `reset()` method didn't need an argument. We know what has to be done to put a circle back in the middle of the screen. But should you write a method to place a circle on any position, you'ld have to pass the desired position as an argument. Like so:

```
class movingCircle {
  Circle c;
  Vec2 speed;

  void setPos(Vec2 pos) {
    c.pos = pos;
  }
}

// somewhere in your application
movingCircle mc;
Vec2 p(0.1, 0.4);
mc.setPos(p);
```

This method has an argument of the type `Vec2`. The argument is used internally to alter the circle's position. It is important to remember you only pass the value of the argument, not its name or the variable itself. There is no established relationship between an method and the variables passed into it.

```
movingCircle mc;
Vec2 p(0.1, 0.4);
mc.setPos(p);     // pass contents of p to mc
p.x += 0.1;       // altering p does not alter the position in mc!!!
mc.setPos(p);     // call setPos again to pass the new contents to mc
```

Time for Action

Add the method `setPos` to your exercise. With it, try something new in the application: every time you press F1, a new random position should be assigned to every circle. Add a method `setRadius` with a float argument. This method should change the radius of the circle. Change the radius along with the the position when you press F1. (Remember to use only small values, like a 0.01 to 0.2 range.)

9.3 More arguments

Methods are not limited to a single argument. You can add more as long as they're separated with comma's.

```
class movingCircle {
  Circle c    ;
  Vec2   speed;

  void init(float r, Vec2 pos, Vec2 speed) {
    c.r     = r    ;
    c.pos   = pos  ;
    T.speed = speed;
  }
}

// somewhere in your application
movingCircle mc;
Vec2 p(0.1, 0.4);
mc.init(0.1, p, Vec2(0.3, -0.5));
```

Again, the example introduces a few new concepts:

T What if the name of an argument is the same as the name of a class variable? In the example above there is an argument called 'speed', but also a variable called speed. How will the compiler know which is which, when you just type speed. Well, as a rule the most local declaration is used, which in this case is the argument. To indicate you want to use the class variable, you can use `T`. Again in the example, the value of the argument 'speed' is passed to the class variable 'speed'.

Passing arguments When the init method is used, a predefined `Vec2` is used to pass the position. But speed also needs a `Vec2`. In this case, a new Vec2 is created directly inside the braces and passed as an argument. Both options are valid, but the first one is generally used when you still need the object p after passing it to the function. This would not be possible with the second `Vec2` because it does not have a name.

Time for Action

And in init method to your class and use it.

9.4 Methods with a Result

In the beginning of this chapter, I mentioned you should never use class variables directly, outside a class. You've just learned how to alter a variable through a method with an argument, but what if you wanted to retrieve the value of the variable? The answer is simple. Use the method result:

```
class movingCircle {
  Circle c;
  Vec2 speed;

  void init(float r, Vec2 pos, Vec2 speed) {
    c.r          = r    ;
    c.pos        = pos  ;
    this->speed = speed;
  }

  float getRadius() {
    return c.r;
  }
}

// somewhere in your application
movingCircle mc;
mc.init(0.1, p, Vec2(0.3, -0.5));
float radius = mc.getRadius();
```

At the end of this example, the float radius will have a value of 0.1. This is how you create a method with a result:

- ❏ Use the result type instead of the keyword void. In this case we want to retrieve a radius from the circle. A radius is a float, which means the result type will also be a float.
- ❏ The method contents should end with the keyword `return`, followed by the value that should be returned. This value will be passed to the caller.

Time for Action

Add this method to your class. Add code to your application to show the radius of a circle on the screen.

9.5 Methods with Arguments and Result

One last possibility is the combination of arguments with a result. One common pitfall: although a method can have many arguments, it can only have one result.

The next example implements a `move` method. This method will move a circle by passing a value as an argument. This value will be added to the currrent position, but only as long as the result would not be outside of the screen. By returning a bool as a result, we can let the application know if movement succeeded.

```
class movingCircle {
  Circle c    ;
  Vec2   speed;

  void init(float r, Vec2 pos, Vec2 speed) {
    c.r         = r    ;
    c.pos       = pos  ;
    this->speed = speed;
  }

  float getRadius() {
    return c.r;
  }

  bool move(Vec2 pos) {
    if(Cuts(c.pos + pos, D.viewRect())) {
      c.pos += pos;
      return true;
    } else {
      return false;
    }
  }
}

// somewhere in your application
movingCircle mc;
Vec2 p(0, 0);
mc.init(0.1, p, Vec2(0.3, -0.5));
if( !mc.move(mc.speed * Time.d()) ) {
  // do something when failed
}
```

Time for Action

Add the move method to your class. The application must use this method to move the circle. This method indicates wether or not the movement was a success, so you can remove the application code to keep the circle within the screen boundaries. If the method returns false, simple call the `reset` method to put the circle back in the middle. It is still possible to simplify move even further. We already know movement will always be the same. So we don't actually need an argument at all! The movement can be done with the information available in the class itself. (speed and the time delta)

9.6 The solution

The complete class and app for this chapter is listed below for reference. Please compare this with your own result.

De class movingCircle:

```
class movingCircle
{
   Circle c    ;
   Vec2   speed;
   Color  color;

   void create(float radius, Vec2 pos, Vec2 speed, Color color)
   {
      c.r     = radius;
      c.pos   = pos   ;
      T.speed = speed ;
      T.color = color ;
   }

   bool move()
   {
      Vec2 newPos = c.pos + speed * Time.d();
      if(Cuts(newPos, D.viewRect()))
      {
         c.pos = newPos;
         return true;
      } else return false;
   }

   Vec2 getPos()
   {
      return c.pos;
   }

   void setPos(Vec2 pos)
   {
      c.pos = pos;
   }

   void reset()
   {
      c.pos = 0;
   }

   void draw()
   {
      c.draw(color);
   }
}
```

The application:

```
Memc<movingCircle> circles;

void InitPre()
{
   EE_INIT();
}

bool Init()
{
   return true;
}

void Shut() {}

bool Update()
{
   if(Kb.bp(KB_ESC)) return false;

   if(Kb.bp(KB_SPACE))
   {
      circles.New().create(RandomF(0.05, 0.1),
                           Vec2(0),
                           Vec2(RandomF(-1, 1), RandomF(-1, 1)),
                           Color(Random(255), Random(255), Random(255))
                           );
   }

   if(Kb.bp(KB_F1))
   {
      REPA(circles)
      {
         circles[i].setPos(Vec2(RandomF(-D.w(), D.w()), RandomF(-D.h(),
            D.h())));
      }
   }
```

```

    REPA(circles)
    {
        if(!circles[i].move()) circles[i].reset();
    }

    return true;
}

void Draw()
{
    D.clear(BLACK);

    REPA(circles)
    {
        circles[i].draw();
    }
}
```

Chapter 10

More fun with Classes

By now, you probably have a pretty good idea how to create a class. But there's no single right way to create and use classes. Different programmers use different approaches. Mostly, they're all valid and the choices are based on personal preferences. Still, when you start working with classes, the possibilities can be a bit daunting.

In this chapter, we'll examine one approach which is very useful when working with Esenthel.

10.1 Related ideas

Every 'idea' in your application should be a class. This idea can be the score logic, the database connection, the player, a certain type of monster and so on. A class is something you create objects from, so things like a player, a monster or a window do seems like logical candidates. And they are. But just as well can a class be a more abstract idea, like a scoring system, everything related to the network or a database connection. Or a class could just be responsible for retrieving and updating the data from one table in a database.

Everything that is related to that idea, should be part of the class. Take a score class, for example. The following concepts clearly belong together:

- ❏ a variable to store the current score
- ❏ a variable to store the high score
- ❏ a method to change the score

- a method to reset the score
- a method to compare the score with the high score
- a method to retrieve the current score
- ...

When creating a class, it is a good idea to think about the class itself for a moment. What methods could you might need later on? Maybe you won't need a reset method, but there's a good chance you will. Instead of thinking about your application, you should think about the class itself.

Beginning programmers often make the mistake to focus on their application a bit too much. The result of that is a programming session like this:

- Hey, i need a way to store an view the current score. Let's make a class for that.
- (you write a class with a score and set and get methods)
- Great, now i can create a window which shows the score on the screen. Let's make a class for this window.
- (you write a class for the score GUI)
- Oh, let's add a highscore!
- (you alter the score class to store the high score)
- (you start creating a database class to store highscores)
- (you alter the score gui to display a high score)
- Actually, I need a method to add points to my score.
- ....

The problem with this is that within a few minutes, you're working on three or more classes at the same time. Programming is hard enough as it it! When you're trying to keep track of the content and changes of different classes at the same time, you're making everything much harder for yourself.

This is why you should try to focus on the current task: creating the class. Maybe you will implement a few methods you won't even need later on. That's OK! It is much harder when you constantly have to revisit your class because you need something which doesn't exist.

That's rule number one: think about the class, not about the application.

Rule number two is: logic and interface don't mix. It's just a bad idea. We try to make a clear distinction between GUI classes, logic classes, network classes, database classes and so on.

Think about the score class one more time. You could decide that the score will be visible on the screen and implement a score method called 'draw' to do just that. But later on you need to display the score when the game is over, in a slightly different way. Will you create yet another draw method? Or when you have to line up the drawing of score with the drawing of the current level. Will you add the level to the scoring system? Level management could very well deserve its own class.

A far better approach is to create a method to retrieve the score itself. And write classes for a GUI during the game and a GUI during a game over. Those classes will retrieve the current score and decide on how to display it.

```
class score {
  int points = 0;

  void reset() { points = 0;     }
  void get   () { return points; }
  void inc   () { points++;      }
}
score Score; // object

// this class will be used during the game
class overlay {
  void draw() {
    D.text(Vec2(0, 0.9), S + "Score: " + Score.getPoints());
  }
}

// this class draws a gui during game over
class gameOver {
  void draw() {
    D.text(Vec2(0,0), S + "Score: " + Score.getPoints());
  }
}
```

10.2 Setters and Getters

It is generally considered a good idea to use methods to access a class variable. Set methods are used to store a value in a variable, get methods are used to retrieve them. These methods are very easy to create, but will protect against mistakes when your classes become more complex.

For consistency, I recommend you use the following rules to write these methods:

- ❏ The name of the method is equal to the name of the variable, preceded by set or get. (And with camelCaps of course)
- ❏ The method alters the value with the same name as the method.
- ❏ The set method has an argument of the same type as the value. It returns a reference to the class.
- ❏ The get method returns a variable with the same type as the class variable.

Here's an example of a score class with set and get methods for an integer variable called 'points'.

```
class score {
  int points;

  int  getPoints(         ) { return points ; }
  void setPoints(int value) { points = value; }
}
```

Another way to implement setters and getters would be to use a special symbol in the variable name. Most programmers use an underscore. This approach has the added bonus that there is a clear distinction between class variables and local variables, because all class variables start with an underscore.

```
class score {
  int _points;

  int  points(         ) { return _points ; }
  void points(int value) { _points = value; }
}
```

10.3 public and private

In the previous example, it is still possible to change the points variable directly, without using the provided methods. This is frowned upon by most programmers. If set and get methods are provided, they should be the only way to access the class variable. Suppose your class keeps a counter to know how many times the variable has changed. It would be easy to increase this counter every time the setPoints method is called. But the result cannot be trusted as long as you can also change the value without using this method.

Everyone makes mistakes, so it is better to make sure they cannot happen. Which can be done by making the class variables private: only the member functions of the class will be able to access private variables or methods. The methods themselves can be declared public, which allows you to use them from outside the class.

```
class score {
private:
  int points  = 0;
  int counter = 0;

public:
  int getPoints() {
    return points;
  }

  score & setPoints(int points) {
    this->points = points;
    counter++;
    return T;
  }
}
score Score;

// somewhere in your application
Score.points = 3; // won't work
Score.setPoints(3); // does work and increases the counter
```

10.4 Global Objects

Some classes are intended to create lots of objects. Take `Vec2` for example: you will probably need a lot of those in your application. But other classes, like the score class above, are only meant for a single object. In this case it makes sense to construct that object globally, right below the class. A global object has the advantage that it can be used from anywhere in your application.

10.5 Manager Classes

When you create a class for moving circles, because you will need a lot of moving circles in your application, it might be a good idea to create a special class to handle them. Such a 'manager' class will be responsible for creating new moving circles, updating and drawing them, and even delete them when no longer needed.

Suppose you have the following class:

```
class myCircle {
private:
  Circle c;
public:
  void update() {
    // update code here
  }

  void draw() {
    // draw code here
  }
}

class myCircleManager {
private:
  Memx<myCircle> list;

public:
  void createCircle() {
    myCircle & temp = list.New();
    // do something else with the new circle if you like
  }

  void update() {
    // update all circles
    for(int i = 0; i < list.elms(); i++) {
      list[i].update();
    }
  }

  void draw() {
    // draw all circles
    for(int i = 0; i < list.elms(); i++) {
        list[i].draw();
      }
    }
  }
}

myCircleManager MyCircleManager;

// your application
bool Init() {
  MyCircleManager.createCircle();
  return true;
}

bool Update() {
  MyCircleManager.update();
  return true;
}

void Draw() {
  MyCircleManager.draw();
}
```

Time for Action

1. Create the class 'movingCircle'. It should contain a `Circle`, a `Vec2` 'direction' and a float 'speed'.
2. Add a 'create' method to provide the circle with a radius and a position. This method should also assign a random float between -1 and 1 to the direction variable, and a random float between 0.5 and 2 to speed.
3. Add an 'update' method to this class. This method will add the direction, multiplied by `Time.d()` and 'speed', to the circle's position. The update method should also check the new position. Whenever the x or y value is out of bounds (which means it is no longer visible on the screen) you should reverse the sign of that value.
4. Create a 'draw' method to display the circle on the screen.
5. Create a global object of the class 'movingCircle' in your application. Don't forget to add the object's methods and run your application to see the result.
6. Add a new class 'circleManager' which contains a memory container for the class 'movingCircle'.
7. Provide an 'add' method to add a new circle to this container.
8. Provide an 'update' method which calls the update method of every circle in the container.
9. Add a 'draw' method to draw all circles on the screen.
10. Since you only need one object of this class, you should add it in the same file, right below the class declaration with the name 'CM'.
11. Remove the object of the class 'movingCircle' from your application. *(The lines where you used this object should also be erased.)*
12. In the update function of your app, execute the 'add' method of CM whenever the space bar is pressed.
13. Also in this update function, execute the update method of CM. And of course the draw method should be called inside your application's Draw function.
14. Extra: write a 'remove' method for the circleManager. This method should accept a position as an argument. The method should compare this position with the circles in it's container. If the provided position overlaps with a circle, this circle should be erased from the container.

10.6 Working with existing code.

When you need to change code which already exists, there's good news and there's bad news. The good news is that it can save you some work because you don't have to start from scratch. The bad news: as long as you don't understand what's already there, you are bound to make mistakes. It can be hard to start working with code you didn't write yourself, but here's a few suggestions to get you started.

- ❏ Be sure to know where to find what. Examine all classes and find out what they're supposed to do.
- ❏ Check which classes have global objects.
- ❏ Find out which class are management classes and what see which classes they manage.
- ❏ Are there any other classes which clearly belong together?
- ❏ It doesn't hurt to take notes about this. A few notes about the existing classes and how they relate to each other go a long way to gain the needed insight for understanding existing code.

When you're working with classes and you feel you're stuck, ask yourself these question to get on your way:

- ❏ What variables does the class have? What variables do I need to change in the method I am working on. And what information can be used to calculate the result? If this information is already available in the class, just use it. Otherwise you will need to provide one or more arguments to your method.
- ❏ What other methods does the class provide? Can the result of such a method be useful to the method you're working on?
- ❏ When your class needs to alter an object of another class, what methods does that class provide to do what you need? If you can't find any, you may want to add that method first.
- ❏ Will this be a set or get method? In that case you can follow the steps mentioned above to create it.

CHAPTER 11

References

take a look at this code:

```
Vec sum(Vec pos1, Vec pos2) {
  return pos1 + pos2;
}

// someplace else
Vec p1(0.1, 0.3, 0.5);
Vec p2(1.9, 2.7, 0.5);

Vec p3 = som(p1, p2);
```

Although the sum function above will give you the expected result, it is far from optimal. Your application has to do quite a lot of work to calculate the result and store it in `p3`.

You already know that a function doesn't know a thing about what is happening in the rest of your application. In this case that means the function will receive two values, somehow calculates the sum of those values and returns the result to the application.

But how is it able to do so if it doesn't 'see' the reset of the application? How does it know the values of its arguments? And how does it know where to put the result? The answer is quite simple, actually: at the moment the application needs the function `sum` to calculate something, it will copy two values to the function. Afterwards, the result is copied back to the application.

The bad news: all this copying around takes time. Below are the steps that are needed to execute the function `sum`.

❏ copy `p1.x` to `pos1.x`

- ❏ copy `p1.y` to `pos1.y`
- ❏ copy `p1.z` to `pos1.z`
- ❏ copy `p2.x` to `pos2.x`
- ❏ copy `p2.y` to `pos2.y`
- ❏ copy `p2.z` to `pos2.z`
- ❏ reserve some memory for the result
- ❏ add `pos1.x` to `pos2.x` and store the result in `result.x`
- ❏ add `pos1.y` to `pos2.y` and store the result in `result.y`
- ❏ add `pos1.z` to `pos2.z` and store the result in `result.z`
- ❏ copy `result.x` to `p3.x`
- ❏ copy `result.y` to `p3.y`
- ❏ copy `result.z` to `p3.z`

And consider this: this function's arguments are simple vectors! What if it would be containers with thousands of element? Or maybe you will use this function so much it eventually gets called a thousand times each second!

In other words: *When you want speed and performance (and is there really a scenario in which you don't?), avoid unneeded copies in your application.*

11.1 Pass by Reference

So far we always passed values to a method. This is called **pass by value**. We literally pass values to a function or a method. Which means we'll be making a copy of the object containing them.

Another way to pass an object is called **pass by reference**. Instead of copying all values, the memory address of the object is passed. In other words: we'll just tell our function 'Hey, the object you need can be found at this or that location.'

In code, it looks like this:

```
Vec som(Vec & pos1, Vec & pos2) {
  return pos1 + pos2;
}

// someplace else
Vec p1(0.1, 0.3, 0.5);
Vec p2(1.9, 2.7, 0.5);

Vec p3 = som(p1, p2);
```

Compare this to the first code example. The only difference is the & (ampersand) between the argument type and its name. That doesn't seem like much, but let's count the number of steps the processor needs to execute this function:

- ❏ put the memory address of `p1` in `pos1`
- ❏ put the memory address of `p2` in `pos2`
- ❏ reserve some memory for the result
- ❏ add `pos1.x` to `pos2.x` and store the result in `result.x`
- ❏ add `pos1.y` to `pos2.y` and store the result in `result.y`
- ❏ add `pos1.z` to `pos2.z` and store the result in `result.z`
- ❏ copy `result.x` to `p3.x`
- ❏ copy `result.y` to `p3.y`
- ❏ copy `result.z` to `p3.z`

Passing the arguments to the function takes a lot less time when using references. In fact, it is almost always a good idea to pass by reference. The only exception are basic types like `int`, `float` and `bool`. This is because when using basic types, copying the address (32 or 64 bits) takes just as much time as copying the value itself. In some cases it might even be more work! (Consider copying a bool or the address of a bool.)

11.2 Return by Reference

I'm sure you got this warm feeling after reading the previous section. The wonderful world of C++ programming has just revealed some of its secrets to you. You're ready to improve the world and this example even further! Because really, is there any reason not to use a reference as a return value? No, there is not! Look at this:

```
Vec & sum(Vec & pos1, Vec & pos2) {
  return pos1 + pos2;
}

// someplace else
Vec p1(0.1, 0.3, 0.5);
Vec p2(1.9, 2.7, 0.5);

Vec p3 = sum(p1, p2);
```

But unfortunately... it is time to get back to reality. Because `p3` is still a plain `Vec`, not a reference. You still need to copy the result:

- ❏ put the memory address of `p1` in `pos1`
- ❏ put the memory address of `p2` in `pos2`
- ❏ reserve some memory for the result
- ❏ add `pos1.x` to `pos2.x` and store the result in `result.x`
- ❏ add `pos1.y` to `pos2.y` and store the result in `result.y`
- ❏ add `pos1.z` to `pos2.z` and store the result in `result.z`
- ❏ pass the address of `result` to the application
- ❏ copy `result.x` to `p3.x`
- ❏ copy `result.y` to `p3.y`
- ❏ copy `result.z` to `p3.z`

So while you thought you were making your code faster, there is actually an extra step now. Life sucks, right?

But hey, why don't we just declare p3 as a reference instead of a plain `vec`? This way, we can store the address instead of copying the values, right?

Think again. Variables are only valid within the scope they are declared. The result of the function `sum` is implicitly declared when you sum them up with the plus operator. This happens within the function itself. The result will run out of scope when the function is done, freeing all memory of local variables such as the result of the calculation.

In other words: you just passed an invalid address to your application, endangering the world while doing so. It's a good thing the compiler will warn you about this, should you ever try.

So is there no way to use a return by reference? Well, there is. Just not to return a local variable. When you return an object which is actually stored in your class, there isn't a problem. In fact, you've already used this:

```
Memc<Vec> points;
Vec & p = points.New();
p.x = 0.1;
...
```

This is just one example of when references are not only OK, but really needed. First off, the reference returned by new is not local to the method. The whole purpose of a memory container is to maintain a list of objects. Having the memory released when the function goes out of scope would not make any sense.

Also, suppose you *don't* use a reference for p. You would be changing a copy instead of the object in the container. Without a reference, you would be unable to alter the objects inside the container.

Does this mean it can't go wrong? Actually it can, but only if you're really asking for it. For instance when you use a reference after deleting the object it references:

```
Memc<Vec> points;
Vec & p = points.New();
points.clear();
p.x = 0.1; // auch!
```

As a rule, try to use references only directly after creating them. If you reuse them later on, you're asking for trouble.

Time for Action

Reopen the exercise you made at the end of the chapter 10.5. Examine its methods one by one and consider where to use pass by reference instead of pass by value.

CHAPTER 12

Pointers

By know you've learned that references are pretty neat. But before references existed, there were pointers. Just like references, they are used to store a memory address. But while references are hard to use wrong, pointers allow you to abuse them in every possible way. They're the bondage lovers of programming!

For instance: a reference *has* to have a valid address when you create it. And once created, the address can never change. You can't create an empty reference and pass the address later on. Pointers on the other hand, can point to anything or nothing. And you can change the contained address as many times as you like.

In most circumstances, using a reference is a lot safer. When a function asks for reference arguments, you can just provide regular objects. It's hard to do something wrong there. But you might run into problems when you want to return a reference. Take this code for example:

```
class players {
  Memc<Player> list;

  Player &  add(Vec2 & pos, Str & name) {
    Player & p = list.add();
    p.set(pos, name);
    return p;
  }

  Player & findByName(Str & name) {
    FREPA(list) {
      if(Equal(list[i].getName(), name) {
        return list[i];
      }
    }
  }
}
// global object
players Players;

// somewhere in your application
Players.add(Vec2(1,1), "niceGuy");
player & friend = Players.findByName("coolGuy");
```

The **add()** method won't give you problems. The reference arguments will always exist when you pass the to this method. (Either they are real objects, or they are existing references. And references cannot be empty.) The result will be valid too, because we are certain a new object will be made and returned.

Now take a look at the method **findByName()**. This method also returns a reference to a player. As long as the player you're looking for actually exists, there isn't any problem. But what if it doesn't? From what object would you return a reference? Remember, a reference can never be referencing a non existing object. So here are your options:

- ❏ create a a new player: This enables you to return a valid reference to the player with this name. But when you use a method created by someone else, this is not really the behaviour you should expect. A method to find something should not create the object it is looking for. Otherwise, how are you able to find out if something exists or not?
- ❏ return a reference to the first player in the list: This might sound like a good idea. At least you do return a valid reference. But what if you're actually looking for that first player? The method returns the first object in the list, but now you don't know if this is because the object does not exist, or because it is actually the first object.

None of the options you have is a good idea. What we need is a way to have our method tell us "'Hey, I don't have the object you're looking for. Stop buggering me!". Or something slightly more polite.

12.1 Pointers to the Rescue

A pointer *can* point to an object. But it doesn't have to. They're like references without the safeguards. Here's a little idea to familiarize you with the idea:

Suppose you have a room full of lockers. Every locker can hold exactly one item. Now you put a mobile phone in locker 1. After a while, you decide to move the phone to locker 2. But just in case someone is looking for this phone in the first locker, you leave a note in there: 'hey, the mobile phone is stored in locker 2 now'.

Locker 1 now points to locker 2. In fact, we can move the phone around and put it in any locker we like. As long as we update the note in locker 1, everyone is able to get our phone with just one easy step: use the location stored in locker 1.

You might be asking yourself: why on earth don't we just put the phone itself in locker 1 all the time? And you're right. Let us make it more complicated: imagine *every* locker has a phone, except for locker 1. For reasons beyond comprehension, the company owning these lockers wants to make sure all phones are used equally. So when someone uses the first phone, the next person should use the second phone. When the last phone is used, we'll have to use the first one again.

This could get really complicated. Imagine you have to notify everyone in the building that you've made phonecall and used the phone in locker 3. And everyone would have to remember that. Won't it be a lot easier to use a simple procedure?

- ❏ Look in locker 1 for a pointer to another locker.
- ❏ Use the locker found in locker 1.
- ❏ Increase the locker number (pointer) in locker 1.

When all phones are completely discharged, we could even replace the note with the locker ID with an empty note. Everyone would instantly know there's no phone available when the note is empty.

12.2 Pointers 101

12.2.1 Declaring a Pointer

This is how a pointer is declared:

```
int * p1;
int* p2;
int *p3;
Str * textPtr;
Player * playerPtr;
```

As you see, the use of spaces does not matter very much. It is best to choose one notation and stick to it.

12.2.2 Assign an Address.

When you declare a reference, you *have* to assign a value to it. With pointers, you don't really have to do that. But you already know that an declaring an integer without assigning it a value results in a random number. The same happens with pointers. A new pointer points to a random address in memory. This is not wrong, but it will most likely crash your application when you try to use it.

```
int * i; // could point to anything!
```

In most cases you will want to assign an actual address. This can be done by assigning the address of an existing variable. Mind though, you'll need to assign the *address* of the variabel, not the value it contains.

In other words, this is wrong:

```
int i = 42;
int * p = i; // tries to assign the value of i as an address in p
```

The pass the address of a variable, use and ampersand (&). Like this:

```
int   i = 42;
int * p = &i; // declaration and initialisation
int * p;      // declaration only
p       = &i; // assign an address
p       =  i; // auch! wrong!!!
```

12.2.3 Changing a value through its pointer

Rather often you will use a pointer to alter the value it is pointing to. But assigning a value to the pointer itself is wrong. So how do you that? You use an asterisk.

```
int   i = 42;
int * p = i ; // assign the address of i to p
*p      = 43; // assign a value to i through the pointer
```

It is also possible to assign the value of another pointer, like this:

```
*p1 = *p2;
```

12.2.4 Pointer to Null

An unassigned pointer points to random memory. So how do you make an ‘empty’ pointer, pointing to nothing? You assign a ‘null’ address. We also call this a ‘null pointer’.

```
int * p1 = null;
```

You could now rewrite the code at the beginning of this chapter. Remember, the whole problem was that we could not return a name when none was found.

```
class players {
  Memc<Player> list;

  // no problem with references here
  Circle &  add(Vec2 & pos, Str & name) {
    Player & p = list.add();
    p.set(pos, name);
    return p;
  }

  // a pointer is used instead of a reference
  Circle * findByName(Str & name) {
    FREPA(list) {
      if(Equal(list[i].getName(), name) {
        return &list[i]; // notice the ampersand!
      }
    }
    // no player found with this name
    return null;
  }
}
// globaal object
players Players;

// somewhere in your application
Players.add(Vec2(1,1), "niceGuy");
player * friend = Players.findByName("coolGuy");

// check if the player exists
if(friend != null) {
  Greet(friend);
}
```

When no player is found with a certain name, a null pointer will be returned. By checking if the returned value is different from null, we make sure the `Greet(player * p)` function is only executed with a valid player.

NOTE

Some memory containers (like Memb) will move their objects around when the container grows. When there isn't enough memory in the current container, the whole container will be moved to a new memory location. This means that pointers to members of that container become invalid. If you need pointers to stay valid, use a container like Memx. Otherwise, only create pointers local to a method.

12.3 All together

When working with pointers and references, we can use both the symbols & and *. What they mean depends on the situation. Below are all the posibilities.

Suppose these variable exist:

```
int   j   = 43;
int & ref = j ;
int * ptr = &j;
```

We can use these symbols

references and pointers

	reference	pointer
Declaration		`int * i;`
Declaration and Initialisation	`int & i = j;`	`int * i = &j;`
Initialisation		`i = &j;`
Assign value	`i = 42;`	`*i = 42;`
Assign value from variable	`i = j;`	`*i = j;`
Assign value from reference	`i = ref;`	`*i = ref;`
Assign value from pointer	`i = *ptr;`	`*i = *ptr;`
Assign address		`i = &j;`
Assign addres from reference		`i = &ref;`
Assign address from pointer		`i = ptr;`

References are a bit easier to use. But also less flexible. Sometimes you really need a pointer.

Time for Action

1. Create an application with 5 circles, spread out on the bottom of your screen. Create a memory container to hold these circles.
2. When the mouse is on top of a circle, it should slowly move upwards.
3. Create a function with a pointer to the highest circle as a result. Call this function in every update and keep the result in a pointer variable.
4. Draw all circles on the screen. Before drawing a circle, compare its address with the address of the highest circle. When they are the same, draw the circle green, otherwise draw it red.

CHAPTER 13

Constants

13.1 Global Constants

Most applications will use particular values throughout the code. Imagine an application which does a lot of calculations with circles: you will definitely use the number pi a lot. You could calculate pi every time you need it, but that's not a good idea because the outcome will be the same every time. You're making your computer do needless calculations. Instead you could declare a global variable:

```
int pi = 3.1415926;
```

With this variable, you can use pi everywhere in your code. But mistakes happen, and what about this one:

```
int value = 1;
// ... more code ...
if(pi = value) {
  // do something
}
```

The code above won't result in an error. But instead of comparing pi, you assign a new value to pi by mistake. Auch! This mistake is easily made, but it might take a while before you realize why every calculation with pi suddenly has the wrong outcome.

It would be much better if we could prevent such a mistake from being made. After assigning a value to pi, there is no reason why it should change. We need a way to prevent changes made to this variable. That's why most programming languages provide a way to make a variable 'constant'. A constant can never changes after its declaration. And to increase readability, most programmers will always write constants in capitals.

So how do you declare a constant in C++? You precede it with the word `const`. Esenthel makes it even easier by defining the capital `C` as `const`, just like you can write `T` instead of `this`. In Esenthel, you could declare PI as:

```
C PI = 3.1415926;
```

This approach has two advantages:

1. The value of PI can never be changed by mistake.
2. If, for some crazy reason, you *need* to change the value of PI, you only have to change it this one instance. *(Highly unlikely in this case, but it will happen with other constant values in your application. Imagine a constant named ATTACK_RANGE. You probably will want to experiment with that before releasing your game.)*

Note

Because PI is needed in almost every game, Esenthel already declared it for you. And not only PI is defined, but also a few common calculations like PI_2 (half of PI) and PI2 (two times PI).

Time for Action

Write an application with the following constants: playerColor, playerSize, enemyColor and enemySize. The player is a rectangle, the enemies are circles *(it is a very abstract game)*. Draw a player and some enemies on the screen using these constants.

13.2 Const Arguments

There's another situation in which constants are used. Just look at this function:

```
float calculateDistance(Vec2 & pos1, Vec2 & pos2);
```

Suppose you can use this function to calculate the distance between two points. In chapter 11 you read that it is often faster to pass a variable by reference as opposed to passing it by value. But there's a downside on that. In principle, you could alter the values of pos1 and pos2 within this function. And if a value is passed by reference, this would also change those variables in the original location. But imagine you are working in a team and the function `calculateDistance` is written by someone else. If something unexpected happens in your code, you would have to double check the work of your colleague to verify the values are not changed in there. The name of the function seems to indicate there's no reason for that, but mistakes happen. A lot.

It would be better if we can now for sure the values we pass to this function will not be changed by it. That way, we can focus on our own code when there's an error. The solution is simple: we can pass these references as constants:

```
float calculateDistance(C Vec2 & pos1, C Vec2 & pos2);
```

This is much better because:

1. when creating such a function, you will get an error if you try to change pos1 or pos2;
2. the programming which uses this function can be sure the values are not changed in there, without reading your code;
3. you have a strong indication that values *will* be changed inside a function when the reference is not declared as a constant. *(It's either that, or you need to have a serious talk with your fellow developers.)*

Let's agree that, starting right now, you will pass all function arguments as a const reference, unless you have a good reason not to. Your future colleagues will like you a lot more when you do so.

And what is a good reason not to use a constant? Look at the Esenthel `Clamp` function:

```
void Clamp(Vec2 & value, C Vec2 & min, C Vec2 & max);
```

This function will change the first argument when its value is lower than the second or higher than the third. It can be used like this:

```
Vec2 pos = Ms.pos();
Clamp(pos, Vec2(-0.4,-0.4), Vec2(0.4,0.4));
pos.draw(RED);
```

The second and third argument are constant, so the function will never change the minimum or maximum. But the first argument is the one that is meant to change. No const reference there.

Time for Action

- ❏ Search the Engine code for more functions with non-const references. Try to explain why these references are not constant.
- ❏ Write a function 'ClampToScreen' which changes an argument (a Vec2) when it would be outside the screen. Test your function with a simple program. Can you use a const reference?
- ❏ Write a function with a string argument. The function will draw the string on the screen. Create a version with a const reference and one with a non-const reference. Test both versions with existing `Str` variables as well as with string literals. Why won't the non-const version work with string literals?

Chapter 14 Enumerations

14.1 Bad Example

Enumerations (also often called 'enums') allow us to display numbers as text. Take an enemy class for example. The enemy could be a warrior, a rogue or a priest. Depending on that choice, another image would be displayed on the screen. You could use booleans to remember what type of enemy you're dealing with:

```
class enemy {
  bool warrior = false;
  bool rogue   = false;
  bool priest  = false;
  Rect r;

  void setWarrior() {
    warrior = true ;
    rogue   = false;
    priest  = false;
  }

  // the methods setRogue and setPriest are similar
  // ...

  void draw() {
    if      (warrior) Images(=== warriorImage ===).draw(r);
    else if(rogue   ) Images(=== rogueImage   ===).draw(r);
    else if(priest  ) Images(=== priestImage  ===).draw(r);
  }
}
```

Even though this works, it isn't very efficient. Right now, we only got 3 types of enemies, but the more possibilities you have, the more variables you need to change when selecting a type. It would be much better to use a single variable.

14.2 Slightly less Bad Example

You could decide to give a warior the number 0, a rogue 1 and a priest 2. This makes your code a lot easier.

```
class enemy {
  int type = -1;
  Rect r;

  void setType(int type) {
    T.type = type;
  }

  void draw() {
    switch(type) {
      case 0: Images(=== warriorImage ===).draw(r); break;
      case 1: Images(=== rogueImage   ===).draw(r); break;
      case 2: Images(=== priestImage  ===).draw(r); break;
    }
  }
}
```

Although this is better than the first version, you will mess up sooner or later and use the wrong number somewhere. Or maybe you will use a number for which no class exists.

14.3 Enumeration time!

The solution for this problem are enumerations. These are lists of words. Internally the first word is equal to zero and every word after that is a higher number. The interesting part is that you can use these words in your code, but the application will use numbers internally.

```
enum ENEMY_TYPE {
  ET_NONE    ,
  ET_WARRIOR,
  ET_ROGUE   ,
  ET_PRIEST ,
}

class enemy {
  ENEMY_TYPE type = ET_NONE;
  Rect r;

  void setType(ENEMY_TYPE type) {
    T.type = type;
  }

  void draw() {
    switch(type) {
      case ET_WARRIOR: Images(=== warriorImage ===).draw(r); break;
      case ET_ROGUE   : Images(=== rogueImage    ===).draw(r); break;
      case ET_PRIEST  : Images(=== priestImage   ===).draw(r); break;
    }
  }
}
```

Values like `ET_WARRIOR` or `ET_ROGUE` can be used anywhere in your code. The method `setType` won't accept a type of enemy which doesn't exists. And at every moment you will have clear idea of what you're doing.

The values of an enum are often capitals. This is just a convention, but it's a good one. You can instantly see that this is not a variable. And just like with constants (also often written in capitals), the value of an enum cannot be changed. Something like `ET_WARRIOR = 3` is not possible.

Starting the enum value with `ET_` is also a matter of choice. I like to use the first characters of the enumeration in its value name because it makes them easier to remember. "ENEMY_-TYPE" can be remembered as ET. The autocomplete of the editor will help you by showing your options as soon as you've typed `ET_`.

Note

Esenthel also provides an *enumeration editor*. This is not needed for simple enumerations, but we will use it later in this course because the enumerations defined in there are also available in the world editor.

Chapter 15 Application States

More often than not, an application will provide different "stages". On stage could be the moment when you're in the game lobby. Another could be the actual game, or the stage where you pick your character. There won't be much shared code code in the update and draw functions in all those states.

Esenthel provides one application state by default. It consists out of the functions `Init()`, `Shut()`, `Update()` and `Draw()`. When a state becomes active, its `Init()` function is executed. After that, the `Update()` and `Draw()` function are constantly alternated until you close the application or switch to a different state. At the moment, the `Shut()` function is called.

15.1 Intro

Every state should be written in its own file. This could be the intro state:

```
bool InitIntro() {return true;}

void ShutIntro() {}

bool UpdateIntro()
{
   if(StateActive.time()>3 || Kb.bp(KB_ESC)) {
      StateMenu.set(1.0);
   }
   return true;
}

void DrawIntro()
```

```
{
   D.clear(BLACK);
   D.text (0, 0, "Intro");
}

State StateIntro(UpdateIntro, DrawIntro, InitIntro, ShutIntro);
```

This looks a lot like the default state we used so far. The word 'Intro' is added before all functions to keep it comprehensable. The actual stated is constructed in the last line:

```
State StateIntro(UpdateIntro, DrawIntro, InitIntro, ShutIntro);
```

The state is given a name, and all functions are added to the state. The only thing which is not present is an `InitPre()` function. That one is not part of a state. It is the first function the application will call when started.

Take a look at the constructor of the `State` class:

```
State(Bool (*update)(), void (*draw)(), Bool (*init)()=NULL, void
    (*shut)()=NULL);
```

Does the asterisk ring a bell? Indeed, it's a pointer. A function pointer to be more precise. The constructor expects us to provide it with the functions that should be ran in this state. Also, the init and shut functions are optional. (They have default value which is a null pointer.) If you don't need to initialize or cleanup, you may leave this empty.

Note

When a function argument ends with `=NULL`, it is an optional argument.

15.2 Menu

The code above also mentions another state: 'stateMenu':

```
   if(StateActive.time()>3 || Kb.bp(KB_ESC)) {
      StateMenu.set(1.0);
   }
```

In other words: we will wait 3 seconds in this state, or until the user presses escape. At that time, another application state will become active, with a crossfade of 1 second between them. The menu state could be something like this:

Deze nieuwe state zou er zo kunnen uitzien:

```
bool InitMenu() {return true;}
void ShutMenu() {}

bool UpdateMenu()
{
   if(Kb.bp(KB_ESC))return false;
   if(Kb.bp(KB_ENTER))StateGame.set(0.5);
   return true;
}

void DrawMenu()
{
   D.clear(GREY);
   D.text (0,  0  , "Menu");
   D.text (0, -0.3, "Press Enter to start the game");
   D.text (0, -0.5, "Press Escape to exit");
}

State StateMenu(UpdateMenu, DrawMenu, InitMenu, ShutMenu);
```

This state is very similar to the previous one. But this time we can use the Enter key to enter the actual game.

15.3 Game

You can use this code for `StateGame`. Again, add this code in a separate file.

```
bool InitGame() {return true;}
void ShutGame() {}

bool UpdateGame()
{
   if(Kb.bp(KB_ESC))StateMenu.set(1.0);
   return true;
}

void DrawGame()
{
   D.clear(TURQ);
   D.text (0, 0, "Game");
}

State StateGame(UpdateGame, DrawGame, InitGame, ShutGame);
```

You will switch back to `StateMenu` by pressing escape. In a real game you will lot of other code in the update function.

15.4 Default State

Now we still have to start the application. We got all the states we need, but we still have to activate `StateIntro`. This will be done in the `Init()` function off the application. We'll instantly switch to `StateIntro` at that point. This means the functions `Update()` and `Draw()` will never be used in this application.

```
void InitPre()
{
   EE_INIT();
}

bool Init()
{
   StateIntro.set();
   return true;
}

void Shut() {}
bool Update() {return false;} // unused
void Draw  () {             } // unused
```

Time for Action

Use the code in this chapter to create an application which switches between all states. Create a separate file for every state.

Part III: Tetris

You will create a clone of the famous Tetris game. You will learn to develop a project step by step without losing sight of the whole.

Chapter 16

Introduction

In the previous chapters you have seen everything you need to create a simple 2D game. But how do you put a large project together in an orderly way? There is really no simple answer to that. You learn by practice, and not everyone agrees on the best method. Still, there are some rules that may make it easier for sure. And when you work in a group, the lead programmer will usually impose some rules that everyone has to follow. These are not necessarily good or bad rules, but they work as long as everyone follows them.

In this part of the course you will create a clone of the famous Tetris game. You will learn to develop a project step by step without losing sight of the whole.

Note

Tetris consists of blocks which in turn consist of squares. When we talk about blocks in this course, we mean the entire block, not the squares it is made of. When a square is mentioned, we're discussing the squares that make up a block.

16.1 Setup

Open the 'Tetris_start' project. In it, you will find the graphics, sounds and fonts that we will use. A blank app 'Tetris' is also provided, but we are not going to use it just yet.

1. Create a library at the highest level of the explorer. You do this by right clicking and choosing 'new library'. Name this library 'Tetris parts'. A library is a green folder. The code in a library can be used from any application within your project, just like the library 'Esenthel Engine' which is always present.

2. Create a new application (blue folder). Call it ‘square tester’.
3. in the application ‘square tester’, create a code file ‘main’.
4. In the library ‘Tetris parts’, create a new folder (yellow) called ‘definitions’.
5. Mark ‘square tester’ as the active application.

Copy the code in ‘Tetris/initState’ to ‘square tester/main’. Remove this line:

```
D.full(true);
```

This makes it easier to terminate your application when something goes wrong.

16.2 Constants

In the previous chapter you learned about constants. It is a good idea to create some important constants before you start on the actual code. You can use them anywhere in your code and easily adjust their values if necessary. Create a new code file ‘constants’ in the folder ‘Tetris parts/definitions’.

To be able to change the name of the app later on, create a constant `Str` with a provisional name.

```
C Str APP_NAME = "Tetris";
```

The size of the standard application window is not ideal for this game. A size in pixels is required for this. Declare a constant `int` for this purpose.

```
// The window size on the screen, in pixels
C int WINDOW_WIDTH  = 900;
C int WINDOW_HEIGHT = 800;
```

Tetris consists of rows and columns. we also define these:

```
// This impacts the playing field
C int SQUARES_PER_ROW = 10;
C int ROWS            = 15;
```

It is also possible to add a few constants for the scoring system. There is a fixed number of levels, and we know how much points will be rewarded for a line and a level.

```
// The scoring system uses these
C int POINTS_PER_LINE  =  525;
C int POINTS_PER_LEVEL = 6300;
C int NUM_LEVELS       =    5;
```

The speed of the game goes up with each level. This change can also be defined as a constant.

```
// The speed will increase every level
C float INITIAL_SPEED = 1.0;
C float SPEED_CHANGE  = 0.1;
```

When playing tetris, there is a short period after moving a block down in which you are able to move it sidewards. This period has to be defined.

```
// The time a block can be slided to the side
// When it hits bottom
C float SLIDE_TIME = 0.25;
```

We also have to determine the size of the playing field. This is done by defining the position of the lower left corner, and defining the size of the rectangle containing the playing field.

```
// The area reserved for the playing field
C Vec2 GAMEAREA     (-0.8, -0.8);
C Vec2 GAMEAREA_SIZE( 1.0,  1.4);
```

A new block will always appear at the top of the screen. This position can be deducted from information we already have: SQUARES_PER_ROW and ROWS. In addition, there is a waiting position, top right of the playing field. You might notice we are not using `Vec2`, but `VecI2`. This is a vector which fits only integers. We do not want Tetris blocks midway between two positions, so there is no need for floats here.

```
// Position for the current and next block
C VecI2 STARTPOS(SQUARES_PER_ROW / 2, ROWS - 1);
C VecI2 WAITPOS (SQUARES_PER_ROW + 4, ROWS - 3);
```

Now it is possible to calculate the size of a square with the information we already have. This has the advantage that we can modify the foregoing constants later, the size of a square being automatically adjusted.

```
// The size of a square
C float SQUARE_SIZE = GAMEAREA_SIZE.x / SQUARES_PER_ROW;
```

Note

Of course it will rarely happen that you precisely know which constants are needed when you're just starting out on a project. In practice you will usually add a lot of constants while you are working on your project.

16.3 Enums

Create the code file 'enumerations' in 'Tetris parts/definitions'. Add two enumerations that will be useful in your project. Firstly, there is the block type. Tetris has square blocks, T-blocks and so on. A list might look like this:

```
enum BLOCK_TYPE
{
   BT_SQUARE     ,
   BT_T          ,
   BT_L          ,
   BT_BACKWARDS_L,
   BT_STRAIGHT   ,
   BT_S          ,
   BT_BACKWARDS_S,
   BT_NUM        , // number of block types used in the game
   BT_BACKGROUND , // special case, only for background
   BT_WALL       ,
}
```

The last three values deserve extra attention. The value 'BT_NUM' is useful because the number which represents that value is equal to the highest value + 1. This makes it easy to use a random function. Since the result of a random function does not include the maximum value, we will be able to use this kind of code later on in our application:

```
blockType type = Random(BT_NUM);
```

So why is BT_NUM not the last value in the list? Well, the last two values are special cases. There will be special squares to draw the background and the borders of the game. And the color of such a square is determined by the block type. Because we do not want to use those types in the actual game, BT _NUM is but before these values.

A second enumeration is used to determine the possible directions in which a block can move. A block can not upwards, only to the left, the right, or down. Blocks which are already down, do not have a direction anymore.

```
enum DIRECTION
{
   D_LEFT ,
   D_RIGHT,
   D_DOWN ,
   D_NONE ,
}
```

Chapter 17

Objects

In Tetris you think mainly of blocks as the main item you move around. The fact that blocks exist of squares is ancillary to the player, but it is important for the developer. To check on possible movements all squares have to be evaluated, not the blocks on their own. And when a block reaches the bottom it becomes a part of a lot of squares, which can be removed row-by-row, regardless of the shape of the original block.

17.1 Squares

Create a folder called ‘objects’ in the library ‘Tetris parts’. In it, create a code file ‘square’. This file should contain a class `square` in which we describe a square.

A square must have a position. Because the playing field in Tetris is a grid (we can’t put a square anywhere we want!), we will use another `VecI2` to store integer positions only. Furthermore, the square must have a `block` type to determine in what color it should be drawn. We have already defined the types of blocks in the ‘enumerations’ file.

The class `square` will also need some methods. We provide a `create` method to set the position and type of the block. In addition, we need a `move` method to move the block in a certain direction. (That direction will be the second enum we declared.) We also need a method to retrieve the current position, and a method to change the block’s position directly. And finally, we want a method to draw a square on the screen.

The class looks like this:

```
class square
{
private:
   VecI2      pos ;
   BLOCK_TYPE type;

public:
   void create (C VecI2 & pos, BLOCK_TYPE type) {}
   void move (DIRECTION dir) {}
   VecI2 getPos(              ) C { }
   void setPos (C VecI2 & pos)   { }
   void draw   (              ) C { }
}
```

Add the above code to your project.

17.1.1 Square Tester

You'd expect this is when we add the content of these methods. Instead, we will work on the application 'square tester'. You know you should test your code very frequently during the development of a program. But in a major project that is difficult. We have to write lots of classes before you can actually run the application.

That's why we use test programs. These will be written to test a specific class. In this case, we make sure that all methods of the class `square` can be tested. The code for 'square tester' could look like this:

```

Memc<square> squares;

void InitPre()
{
   EE_INIT();
}

bool Init()
{
   // Here you create the test function, with different
   // block types.
   squares.New().create(VecI2( 2,  2), BT_S          );
   squares.New().create(VecI2( 4,  2), BT_T          );
   squares.New().create(VecI2( 6,  7), BT_L          );
   squares.New().create(VecI2(10, 12), BT_BACKWARDS_S);
   squares.New().create(VecI2( 8,  8), BT_BACKWARDS_L);
   squares.New().create(VecI2( 5,  1), BT_SQUARE     );
   return true;
}
```

```
void Shut() {}

bool Update()
{
   if(Kb.bp(KB_ESC)) return false;

   // We declare a direction and control the arrow keys
   DIRECTION d = D_NONE;
   if(Kb.bp(KB_DOWN )) d = D_DOWN ;
   if(Kb.bp(KB_LEFT )) d = D_LEFT ;
   if(Kb.bp(KB_RIGHT)) d = D_RIGHT;

   // Next we move all the blocks in this direction. When
   // no key is pressed, the squares should not move.
   REPA(squares)
   {
      squares[i].move(d);
   }

   // The methods getPos and setPos should also be tested. We are doing
       that
   // when the spacebar is pressed. We retrieve the current position
   // of each square and change the vertical value. The altered position
   // will be put back into the square.
   if(Kb.bp(KB_SPACE))
   {
      REPA(squares)
      {
         VecI2 pos = squares[i].getPos();
         pos.y += 4;
         squares[i].setPos(pos);
      }
   }

   return true;
}

void Draw()
{
   D.clear(BLACK);

   // Here we test the draw function of each square.
   REPA(squares)
   {
      squares[i].draw();
   }
}
```

NOTE

You have often many ways to write a test program. The important thing is that you test as many class methods in the easiest way possible. This minimizes the chance of errors later on.

17.1.2 Create and Draw

The Create method of square is rather straightforward. You have to make sure the arguments are stored in the class variables. Complete this method on your own.

Slightly more difficult is the draw function. Let's sum up what this method should do.

1. Depending on the BLOCK_TYPE, a color should be set.
2. The position of the square is the position in the grid. We need to calculate the actual position on the screen.
3. We have to show an image on the screen, at the calculated position.

The color will be determined with a switch statement. We declare a variable of the type `Color` and assign the value BLACK. Next, this is changed to the desired color for each type:

```
Color color(BLACK);

switch(type)
{
    case BT_SQUARE      : color = RED   ; break;
    case BT_T           : color = PURPLE; break;
    case BT_L           : color = GREY  ; break;
    case BT_BACKWARDS_L: color = BLUE  ; break;
    case BT_STRAIGHT    : color = GREEN ; break;
    case BT_S           : color = PINK  ; break;
    case BT_BACKWARDS_S: color = YELLOW; break;
    case BT_BACKGROUND : color = Color(50, 50, 50) ; break;
    case BT_WALL        : color = WHITE ; break;
}
```

Next we have to determine the position. The constant GAMEAREA is the lower left corner of the playing field. All positions are calculated from that point. The bottom left corner of the square at the position (0,0) would therefore have to be equal to the lower left corner of the playing field. Therefore:

```
Vec2 screenpos = GAMEAREA;
```

Suppose you want a square at position (1,0). The bottom left corner of the square is equal to that of the playing field, plus the width of one square. For all other positions, the same applies: you multiply the grid position with the size of the square:

```
Vec2 screenpos = GAMEAREA + (pos * SQUARE_SIZE);
```

The upper right corner of the square is exactly one square further. We can create a rectangle to draw on the screen in the following way:

```
Vec2 screenpos = GAMEAREA + (pos * SQUARE_SIZE);
Rect r(screenpos, screenpos + SQUARE_SIZE);
```

To draw the actual image on the screen, use the following code:

```
Images(=== tetris square ===).draw(color, TRANSPARENT, r);
```

17.1.3 Test

At this time the square tester should show the blocks on the screen, even though they can not move yet. Run the tester to confirm this.

The functions move, getPos and setPos are not done yet, but you can surely complete those on your own. Afterwards, verify whether all methods are correct by running the square tester.

17.2 Blocks

Now add a code file 'block' in the folder 'Tetris parts/objects'. Create the class `block` in that file. Just like a square, a block has a position and a type. But in addition, a block is a bunch of squares. So you need a container for squares inside a block.

And then there are methods. eeClassblock needs a create method, just like a square. And we also provide a second create method with other arguments to copy an existing block.

We also need methods to move, rotate and draw a block. And finally we need a method to retrieve the type of a block and a method which returns the list of squares inside the block. These methods only contain a return statement. Because your test application will not work as long as these methods don't contain a return statement, they are already present in the code below. The result is this class:

```
class block
{
private:
   VecI2       pos    ;
   BLOCK_TYPE  type   ;
   Mems<square> squares;

public:
   void create(C VecI2 & pos   , BLOCK_TYPE type) { }
   void create(C block & other                  ) { }

   void move   (DIRECTION dir)   { }
   void rotate(              )   { }
   void draw   (             ) C { }

   BLOCK_TYPE       getType   () C { return type   ; }
   C Mems<square> & getSquares() C { return squares; }
}
```

Add this code in the file 'block' that you created before. Now create a new application 'block tester'. Add code in this application to test a block. This time you do not need a container. Just declare a block and use the create, move, rotate and draw methods. The extra create method and the methods `getSquares()` and {eeFuncgetType() should not be used yet.

17.2.1 Add Squares

Before you start with the create and draw methods, create some 'helper' functions to make it easier on yourself. These methods are in the class `block`, but they are private. The first function is `makeSBlock`. Which looks like this:

```
void makeSBlock()
{
  //     [0][1]
  //  [3][2]

  squares.New().create(VecI2(pos.x    , pos.y    ), BT_S);
  squares.New().create(VecI2(pos.x + 1, pos.y    ), BT_S);
  squares.New().create(VecI2(pos.x    , pos.y - 1), BT_S);
  squares.New().create(VecI2(pos.x - 1, pos.y - 1), BT_S);
}
```

Each block consists of 4 squares. The first square will have the same position as the block. This is important to properly rotate the block later. The other squares should get a position that is derived from the first position. You can look at the diagram in the comments to get a better picture of the positions. The second argument (BT_S) indicates the type of the block. It is important for drawing the square in the right color. You can now create your own functions for the other blocks. Below are the declarations with a diagram for each block.

```
void makeSquareBlock()
{
  // [0][2]
  // [1][3]
}

void makeTBlock()
{
  //    [1]
  // [2][0][3]
}

void makeLBlock()
{
  // [2]
  // [1]
  // [0][3]
}

void makeBackwardsLBlock()
{
  //    [2]
  //    [1]
  // [3][0]
}

void makeStraightBlock()
{
  // [2]
  // [1]
  // [0]
  // [3]
}

void makeBackwardsSBlock()
{
  // [1][0]
  //    [2][3]
}
```

We now create an additional private function **setupSquares()**. In it, you first empty the container **squares**. Next, depending on the eeFunc BLOCK_type, you call the appropriate function. You can start from this example and complete it yourself.

```
void setupSquares()
{
   squares.clear();

   switch(type)
  {
      case BT_SQUARE: makeSquareBlock(); break;
```

```
        // create the remaining code your self
    }
}
```

17.2.2 Create and Draw

The `create` method is easy to complete. Assign values to `pos` and `Type`. Then call the method `setupSquares`. The second create method is a concept we haven't used before. Here we get a reference to another block as an argument. It is our intention to make a copy of the other block. We do this by copying over each variable from the other block. Because this is the first time you have to create such a method, the complete code is written below. Add this method to your project.

```
void create(C block & other)
{
   T.pos  = other.pos ;
   T.type = other.type;

   squares.clear();
   FREPA(other.squares)
   {
      squares.New().create(other.squares[i].getPos(), other.type);
   }
}
```

The `draw` method is easy enough. You should write it yourself. To draw a block on the screen you just need to call the draw method of all the squares in the container.

At this moment it is already possible to run a test. The `create` and `draw` methods already work, so you should check if they do. Run the program with all possible block types and correct your code if something seems off.

17.2.3 Move and Rotate

Now to get your block moving. The `Move` method takes a direction as an argument. You should use a switch to, depending on the direction, adjust the x or y position. (Note: This position is the grid position, not the position on the screen. Therefore it should be increased by one. Using a time delta is not needed here.)

Once you have adjusted the position of the block, you must also pass the direction to all squares, so that they will be moved also. Add the necessary code to the `Move` method.

The rotate method is slightly more difficult. We need to rotate all the squares around the position of the block. This works best if the position of the block is (0,0), because rotating around another point is much more complicated. However, the block is not likely to be at that position. Therefore, we will first subtract the position of the block from the position of the square. After that, we exchange the x and y position of the square, and we add the position of the block back to the position of the square. Finally, we put the new position back into the square. The code looks like this:

```
void rotate()
{
   FREPA(squares)
  {
      VecI2 pos = squares[i].getPos();
      pos -= T.pos;
      VecI2 newPos(-pos.y, pos.x);
      newPos += T.pos;
      squares[i].setPos(newPos);
  }
}
```

Now run your test application again. You should now be able to move and rotate a block. Test with different starting positions to make sure it always works.

17.3 Pile

When a block in Tetris reaches the bottom, the block is no more: all the squares of the block will be added to another container: the 'pile'. This 'Pile' class also needs a memory container to hold squares. It also needs methods to simplify the interaction with this pile.

```
class pile
{
private:
   Memx<square> list;

   bool canMove   (C square & s,  DIRECTION dir) C {}
   void removeRow(int row                     ) C {}

public:
   void init() {}

   bool collides(C block & b, DIRECTION dir) C {}
   void add     (C block & b               )   {}

   int checkLines()   {}
   void draw     () C {}
}

pile Pile;
```

It is also possible to create a test application for this class. In this app, you can manually add a few blocks to the pile and show this pile on the screen. The `collides` method can be tested by creating another block in the application and moving it. This is an example of a possible test:

```
if(Kb.bp(KB_DOWN) && !Pile.collides(myBlock, D_DOWN)) {
  myBlock.move(D_DOWN);
}
```

The `check` method can be tested by calling it on pressing the space bar. Of course you first need to add enough blocks to the pile to form a solid line.

17.3.1 Init and Draw

The methods `init` and `draw` are easy. Write them on your own. The init method must empty the container with the squares. The draw method draws all elements of the container on the screen. If you do not know how to do that, review the chapter on containers.

17.3.2 Add

The `add` method requires a reference to a block as an argument. Its function is to add all squares in that block to the container. Therefore, we provided a method in the class eeClassblock to gain access to all the squares of that block. The class `squares` has a method `create` with arguments for a position and a block type. With that information, you write this method like this:

```
void add(C block & b)
{
   C Mems<square> & squares = b.getSquares();
   REPA(squares)
 {
      list.New().create(squares[i].getPos(), b.getType());
 }
}
```

Now that you've finished this feature, you can already try out the init, add and draw function in your test program.

17.3.3 canMove & collides

The method `canMove` is private. It will only be used inside this class, called from the `collides` method. The method should check if a particular square can move in a certain direction (the second argument).

Note that you should not move the square: only check if the movement is possible. For that reason it is not possible to use the `Move` method of class `square`. *(It is also impossible to do so: precisely to avoid this error, the square is passed as a const reference.)*

The method consists of the following steps:

- ❏ Create a local `VecI2` with the square's position.
- ❏ Move this position in the required direction. (Take another look at the move method in `square` if you have a trouble doing that.)
- ❏ Iterate over all elements in the container with squares and check whether that square is at the same position. If it is, the method should return false.
- ❏ If you reach the end of the container without finding a match, the method should return true. Indeed, if there is already a square would be on the new position, the function was abandoned during the audit. Meaning that the square can move in the requested direction.

The public method `collides` also has a direction argument, but with a const reference to a block. It is this method that will be used in the Tetris application.

Because a `block` has a method to retrieve a reference to the squares in the block, we use that first:

```
C Mems<square> & squares = b.getSquares();
```

The rest of the method will call the method `canMove` for each square. Even when only one square cannot move in the desired direction, the block collides with the pile. In this case the result will be false. Otherwise true is returned. Here's the entire method:

```
bool collides(C block & b, DIRECTION dir) C
{
  // Get squares from this block
  C Mems<square> & squares = b.getSquares();

  // check all of them
  REPA(squares)
{
     if(!canMove(squares[i], dir))
```

```
 {
        return true;
 }
}
  return false;
}
```

Now that you have completed these methods, you can reuse your test application to verify this class.

17.3.4 checkLines & removeRow

Finally, we need to be able to check the pile for full rows and remove them. The method `removeRow` is not that difficult. The argument is the row to remove. So you should once more iterate over all elements in the container. When the `y` position of an element is equal to the argument 'row ', the element should be removed. Otherwise, you have to verify that the `y` is greater than the argument row. If it is, you move the square down with the method `move`.

> **Note**
>
> Because of the way in which the container MemX works, there is no method `remove`. You need to call `removeValid` instead. The differences between the various types of containers will be discussed later.

The method `checkLines` should check for full rows in the pile. If so, those rows are to be removed. The method must return the number of rows deleted, because that is important for the score.

You can copy the code below. Just make sure you understand all the steps.

```
int checkLines()
{
  // Create an array for all rows. Here we provide three additional
  // rows. At the time that the game is done, there will be
  // blocks in the pile that are positioned higher than the actual
  playing field
  int squaresInRow[ROWS + 3];

  // Set all values to zero.
  REPA(squaresInRow) squaresInRow[i] = 0;

  // Iterate the list and raise a row depending on the \verb|y| position
  // of the square.
  REPA(list)
{
     int row = list[i].getPos().y;
```

```
        squaresInRow[row]++;
    }

    // Start with 0 full lines.
    int completedLines = 0;

    // Iterate over all rows.
    REPA(squaresInRow)
    {
        // If the number of squares in this row is equal to the
        // constant SQUARES_PER_ROW, the row is complete.
        if(squaresInRow[i] == SQUARES_PER_ROW)
     {
            // We delete this row, but have to account
            // for the fact that multiple rows can be deleted
            // during this loop. If so, the remaining
            // rows are already moved one position down.
            removeRow(i - completedLines);

            // Finally, we adjust the counter of deleted rows.
            completedLines++;
     }
    }

    // return the number of deleted rows.
    return completedLines;
}
```

17.4 Wall

The last element we need is the 'Wall', the boundary of the playing field. This is not a real element because we can deduce the boundaries of the playing field from the constants we have declared earlier. Yet it is convenient to use the idea of a wall, so that with each movement we can check for collisions with the wall, just as we did with the pile.

Therefore, this class consists of only two methods. A private method `canMove` checks if a square is allowed to move in a certain direction. A public method `collides` checks whether a block would hit a wall if it would be moved. The class looks like this:

```
class wall
{
private:
   bool canMove (C square & s, DIRECTION dir) C {}

public:
   bool collides(C block  & b, DIRECTION dir) C {}
}
```

```
wall Wall;
```

You can surely write this class on your own. The method `collides` is identical to the method in the class `pile`. The method `canMove` is almost the same, but now you don't compare the new position with the content of a pile. Instead, the result is `false` when the x or the y position are smaller than 0. Also if the x position is equal to or greater than the value `SQUARES_PER_ROW`, the result is `false`. In all other cases, the result will be `true`.

You can write a new test program or add the new class to the test application for the pile.

Chapter 18

Interface

In this chapter we will be working on the background and the sound in the game. Again, a test app should be created to test these classes.

18.1 Background

The class for the background is kept simple. We never change anything in the background, so a **create** and method are all you need. We will never need more than one instance of this class. That's why we will create an instance of this class right below its declaration.

The playing field will look like figure 18.1. Use it as a reference during the construction of this class.

To draw the playing field we use squares. This also could be a picture, but the squares are easy to construct and rather fitting for this game. In addition, you need a **Rectangle** to display an image on the position where the next block appears. Finally, we need to know the position where the score and the level should be drawn.

```
class background
{
   Mems<square> squares;
   Rect blockRect;
   Vec2 scorePos;
   Vec2 levelPos;

   void create()
   {
```

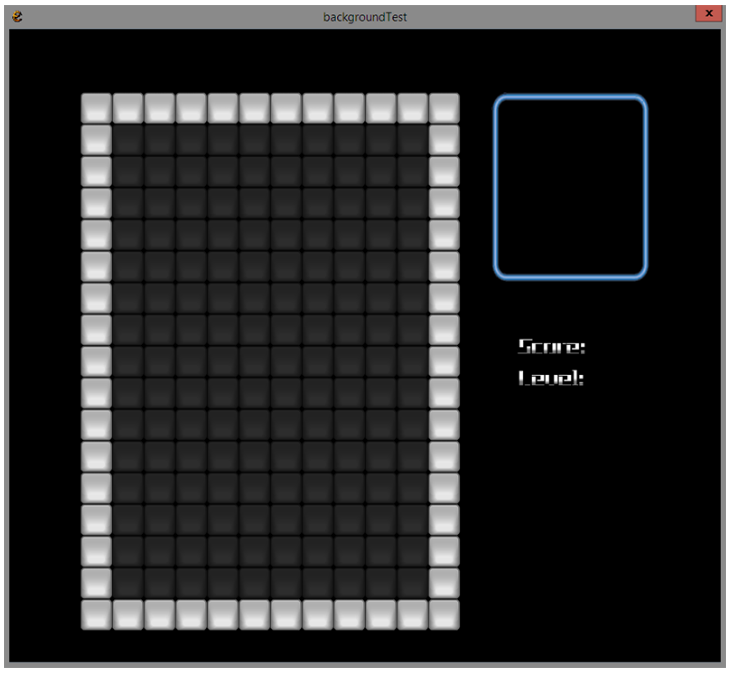

Figure 18.1: The Tetris background

```
    }

    void draw()
    {
    }
}

background Background;
```

In your test app, you can already use the two methods of this class. In addition, add the following line to the InitPre function:

```
D.mode(WINDOW_WIDTH, WINDOW_HEIGHT);
```

This rule ensures that the application window is sized according to our constants. In the previous test program it was not necessary to do so, but with the background we want to see the final result.

18.1.1 Playing Field

As you know, the playing field is a grid. Because we will put a ‘wall’ around the playing field, we don’t start at position 0 but rather at position -1 to add squares. The squares in the field itself will have the type BT_BACKGROUND. The walls have the type BT_WALL. Because of this they will be put on the screen in a different color.

The code to add the necessary squares is shown below. Actually it is not that difficult to understand, but you might have no idea how to start on this. Read this code to gain a full understanding on how this works. Then add it to the create method.

```
for(int x = -1; x <= SQUARES_PER_ROW; x++)
{
  for(int y = -1; y <= ROWS; y++)
  {
    if(x == -1 || y == -1 || x == SQUARES_PER_ROW || y == ROWS)
    {
      squares.New().create(VecI2(x, y), BT_WALL);
    } else {
      squares.New().create(VecI2(x, y), BT_BACKGROUND);
    }
  }
}
```

In the draw method you start with a black background. Next you show all the elements of the 'squares' container. Test your app and verify the result.

18.1.2 Next Block

At the position where the next block appears, we draw an image. We've already made a constant for this position: WAITPOS. But that is a position in the grid. So we should always multiply this position with the size of a square (the constant SQUARE_SIZE). Moreover, we should start at the position of the playing field, which is the constant GAMEAREA.

We can therefore calculate the following position:

```
Vec2 blockPos = GAMEAREA + (WAITPOS * SQUARE_SIZE);
```

But as you know a rectangle requires a minimum and a maximum position. To calculate the minimum, you subtract two squares of the detected position. It would be logical add extra two squares for the maximum position. But later on you will see that this looks less good. This is no problem. You can change this code whenever you like. The last line uses the two new values in order to set the actual rectangle.

```
Vec2 min = blockPos - (2 * SQUARE_SIZE);
Vec2 max = blockPos + (2 * SQUARE_SIZE);
blockRect.set(min, max);
```

Now add code to the draw method to draw the picture 'Tetris_score' using the rectangle you've just created.

18.1.3 Text

You still need two more positions: `scorePos` and `levelPos`. For the position of the score, blockPos will be used as a starting point. Subtract `SQUARE_SIZE` four times of the vertical value. Now you need the position to draw the level. Start from the score position, and subtract another square.

Once you've done that, you are ready to draw the score on the screen. Right now, it will suffice to show a random text. The real score will be added later. Besides, a special font is supplied with this template project. You will find it in gui ⇒ tetrisFont ⇒ tetris Style. If you do not know how to customize the look of text, you can review Chapter 3.4.

Test your application when you're done.

18.2 Sound

The next class is `soundManager`. This one is pretty simple and you surely can write it yourself. There are no create methods required, only methods which play a particular sound. You will write these methods:

1. startMusic: Start the soundtrack in a loop.
2. blip: play the 'blip' sound.
3. score: play the 'rowdone' sound.
4. win: play the 'won' sound.
5. list: play the 'lost' sound.
6. rotate: play the 'rotate' sound.
7. moveDown: playing the 'down' sound.

This class also needs only one instance, so you should create the a `SoundManager` below the class definition. Once done, you can create a little test in which you press keys to play a certain sound. Ensure that all sounds equally loud. If necessary, you can adjust the volume of certain sounds.

CHAPTER 19

Application States

Now that all parts are ready, we start working on the actual application. In it, there are three application states needed: the initState where the program starts, the gameState in which the actual gameplay takes place and the scoreState which shows you the score when the game is over.

Prepare an application 'Tetris' containing a folder 'states'. In that folder, you create three files: initState, gameState and scoreState.

19.1 Init State

In the init state, we provide the regular application functions: **`InitPre`**, **`Init`**, **`Shut`**, **`Update`** and **`Draw`**. You can take these from any program that you made earlier. Next, you provide the functions with the necessary code.

So far, we mostly left the **`InitPre`** function alone. This is the first function that is executed when the application starts. You can use this function to adjust some basic settings. In this case, that will be the name of the program and the size of the window. This can be done by using the constants we declared before.

```
void InitPre()
{
   EE_INIT();
   App.name(APP_NAME);
   D.mode(WINDOW_WIDTH, WINDOW_HEIGHT);
}
```

The init function contains two lines of code. You should call the create method of the object Background and the startMusic method of the object SoundManager. You can certainly write this without an example, right?

We'll skip the Update function for now. But in the Draw function you can go wild for a moment. This is where you create a beautiful startup screen. You can certainly use the image and show some text in the accompanying Tetris fonts. And with some extra effort, it is not too hard to make those move or even change colors. Make sure you also show a text "Push space to start", because that is how we will start the game.

19.2 The Game State

In the file 'gamestate', you add three functions: GameInit, GameUpdate and GameDraw. These functions will be left empty for now, although you can already draw the background if you like. You also need to declare this new application state. Here you can see how this file should look right now:

```
bool GameInit()
{
   return true;
}

bool GameUpdate()
{
   if(Kb.bp(KB_ESC)) return false;
   return true;
}

void GameDraw()
{
   Background.draw();
}

State GameState(GameUpdate, GameDraw, GameInit);
```

Now add some code to the Update function of the initial state: when you press the space bar, your application should switch over to the GameState, with a fade of 0.4 seconds. (If you do not know how to do that, look in Chapter 15.

Test your application to see if the switch works.

19.3 Score State

This application state is closely related to the class score. It is therefore necessary to develop them together. Start with the code for the application state:

```
bool scoreUpdate() {
  return true;
}

void scoreDraw() {}

State ScoreState(scoreUpdate, scoreDraw);
```

Create a separate code file for the class `score`. The framework for this class looks like this:

```
class score
{
private:
   int   points;
   int   level ;
   bool  won   ;
   float speed ;

   void checkWin() {}

public:
   void  init      (          )   {}
   void  addPoints (int value)    {}
   float getSpeed  (          ) C {}
   int   getPoints (          ) C {}
   int   getLevel  (          ) C {}
   bool  hasWon    (          ) C {}

   void  gameIsLost(          )   {}
}

score Score;
```

As you can see, this class contains four values: points, level, won and speed. The latter does not really have anything to do with the score. But because the speed depends on the level, it is convenient to put it together with the rest. For example, the score object may also increase the speed when needed.

19.3.1 Simple Things

We'll do the basic functions first. The `init` method assigns all variables a suitable starting value. You start at level 1, with 0 points. And you have not won yet. The speed is equal to the constant `INITIAL_SPEED`.

The methods `getSpeed`, `getPoints`, `getLevel` and `hasWon` just return the value of the corresponding variable.

Write the contents of these five methods yourself.

19.3.2 checkWin & gameIsLost

In `checkWin` you should check if the current level is greater than the constant `NUM_LEVELS`. If so, then 'won' should be assigned `true`. At this point, you also switch over to the 'score' application state by writing `ScoreState.set(1)` and play a festive sound with the `Sound Manager`.

The method `gameIsLost` is similar, but the check is not necessary. You have lost when a block does not fit in the playing field. The code to check that out does not belong in this class. We will write that someplace else, and at that point this method will be called. In this method, you ensure that 'won' equals to `false` and you also switch to `ScoreState`. And this time you start a less festive sound.

19.3.3 addPoints

This last feature will add points to the score. At that time we will also check whether the next level is reached and if so, adjust the speed. We will also check if the player has won the game, because this is the only moment when such can happen.

Again, add the code below to your application. Make sure you understand all of it. Just typing is not enough!

```
void addPoints(int value)
{
  if(value > 0)
  {
    points += POINTS_PER_LINE * value;
    SoundManager.score();

    if(points >= level * POINTS_PER_LEVEL)
    {
      level++;
      checkWin();
      speed -= SPEED_CHANGE;
    }
  }
}
```

19.3.4 Score State

Now you are ready to add the code for the score state. The framework for that state was already done (otherwise you could not use the state in the `score` class). As you can see, this state only contains an `update` and a `draw` method. Nothing needs to be initialised, so an `init` function is superfluous.

For the draw function you can create something similar to the draw function in the Init state. It's probably best if it resembles that one. But you can create a slightly different version depending on whether you have won or not. (You can use the method `Score.hasWon()` to verify that.) You also want to show the score on the screen, and ask the player if (s)he wants to play another game. For the latter you provide a text on screen, requesting to press y or n.

In the update function you check on the y and n keys. When the player presses y, then you switch back over to the GameState. If n is pressed, you exit the application.

Chapter 20

GameLogic

One class remains, but it is the most important. All parts of the game are ready, but you'll have to bring them together to construct the actual game. This is the purpose of the class **gameLogic**. Its framework looks like this:

```
class gameLogic
{
private:
   // blocks in the game
   block currentBlock;
   block nextBlock   ;

   float forceDownCounter = 0         ;
   float slideCounter     = SLIDE_TIME;

   // to move a block completely down
   bool  toBottom      = false;
   float toBottomTimer = 0.05 ;

   bool canRotate            (C block & b               ) C {}
   bool canMove              (C block & b, DIRECTION dir) C {}
   void handleBottomCollision()   {}
   void changeFocusBlock     ()   {}
   void checkLoss            () C {}
   void handleInput          ()   {}

public:
   void create()
   void update()   {}
   void draw  () C {}
}
gameLogic GameLogic;
```

Let's look at the variables first:

current block This is the block you move during the game.

nextBlock This the next block, which is available on the right hand side.

forceDownCounter When we don't move the block down ourselves, it should go down on its With this timer, we arrange how long that takes.

slideCounter Whenever a block hits the pile, tetris allows you to slide it sideways for a little This means we need a timer for that.

toBottom When the space bar is pressed, the block should move all the way But you have to see it move, so you can't just change its position in one go. With this boolean, we can remember if the block should drop down.

toBottomTimer And that movement also needs a timer for each step.

20.1 The easy methods

20.1.1 CheckLoss

A method you can develop without difficulty is `checkLoss`. This method should check whether the player has lost the game. When should that happen? When a new block appears at the top and that block cannot be moved down, the player has lost. And when is it impossible to move a block down. And when it would collide with the `Pile`.

Pile already has a method `collides`. You can pass the object `currentBlock` and the desired direction to that method. If the method returns false, you can call `Score.gameIsLost()`.

20.1.2 Create

The create method will initialize all variables at the start of a game. The first line should be:

```
Random.randomize();
```

Why do we need that line? Computers have a big problem with random numbers. They're a bit too exact for that. We solve this with the `Random` object, but this object internally has a fixed list of numbers which it will use one by one. Every time you ask for a random number, you just get the next number from the list. Should you start at the beginning of that list every time you run your application, you get the same 'random' numbers every time. This will make your game rather predictable over time. The function `randomize` allows you to start at a random place in the table. This way, another part of the list is used every time.

Note

Should you ever develop software for a casino, you'd better not use the default random functions of your environment. It is not that hard to write a program that, in response to the first few results, looks up where the list has started. At that time you can pretty well predict what the next number in the list will be. As a casino prefers to make a profit, a more complex random library should be used.

The following statements provide two random blocks:

```
currentBlock.create(STARTPOS, (BLOCK_TYPE)Random(BT_NUM));
nextBlock   .create(WAITPOS , (BLOCK_TYPE)Random(BT_NUM));
```

Notice that we use `STARTPOS` and `WAITPOS` to set the positions. The second argument is the block type. We want a random block every time, so the random function is used. The argument of `Random` determines the highest number. But that value is not inclusive: for example, when you write `Random(3)`, the result can be 0, 1 or 2. Not 3. So why do we write `BT_NUM`? For this, you have to go back and look in the file 'enumerations'. `BT_NUM` is the eighth item in the list. The first element is equal to 0, so the eighth element is equal to 7. Therefore, the `Random` function will return a number from 0 to 6. And because an enum is nothing but a name for a number, we can easily convert that number to a `BLOCK_TYPE`. Because that is the type the create function expects.

After these statements, we must assign `forceDownCounter` a value of 0, and `SLIDE_TIME` must be assigned to slideCounter. You are surely able to write those statements yourself. Finally, call the `Pile` and `Score` classes' `init` method.

20.1.3 Draw

The draw function of this class has to draw three elements display: the current block, the block in the waiting area and the pile. Add the statements to do so.

20.2 A bit harder

20.2.1 Can Rotate

We already got the methods to check for collisions with the pile or the game area. Now we need to check whether it is possible to rotate a block. This is why the method `canRotate` is created.

In this method, we first create a new block. We do not want to rotate the actual block, but only check if it would be possible. So we call the create function on the new block, with the current block as an argument. Afterwards, the new block will be rotated.

Now we can check if this new block collides with `Wall` or `Pile`. The second argument is the direction `D_NONE`, because we don't want to check a movement. When one of those methods indicates a collision, the function result is `false`. Otherwise it will be `true`.

20.2.2 Can Move

The `canMove` method enables us to check on the displacement in one go. We have keep an eye on both collisions with the wall as well with the pile. If any of these methods returns `false`, the result should also be `false`. If not, the result is `true`. The arguments of these methods can be passed on to the collide methods of `Wall` and `Pile`.

20.2.3 Change Focus Block

When a block is down, you need to add it to the pile and place a new block on top. The type of the block must be equal to the block in the waiting position. After that, you need a decide what type the next block will be.

You can write this method in three steps:

1. Add the current block to the pile.
2. Call the create method of the current block again. The first argument should be the constant `STARTPOS`. The second argument is the type of the block at the waiting position. (Search in the class `block` for a method which returns the type.)
3. Call the create method of 'nextBlock'. This is identical to the statement you previously wrote in the create function of this class.

20.2.4 Handle Bottom Collision

This method describes what to do if a block hits the pile. Four simple statements are needed:

1. Call the `changeFocusBlock` method.
2. Check whether rows can be removed from the Pile.
3. Pass the number of deleted lines (the result of the previous line) to the `Score` object.
4. check if the game is over with the the `checkLoss` method.

20.2.5 Handle Input

We also need a method which reacts when a key is pressed. This will be the method `handleInput`. Each key that we can press during the game should be treated here. When the down arrow key is pressed, you should first check if this movement is even possible. If so, then the block must be moved down and a sound must be played. Like this:

```
if(Kb.bp(KB_DOWN))
{
  if(canMove(currentBlock, D_DOWN))
  {
    currentBlock.move(D_DOWN);
    SoundManager.blip();
  }
}
```

The code to move a block to the left and to the right is similar. Or at least similar enough to do by yourself!

The 'UP' key is used to rotate a block in tetris. Add code to check if this key is pressed. This time you just need to call the method `canRotate` with the current block. If the result of that method is `true`, you rotate the block and play a sound.

And finally there is the space bar. When pressing the space bar the current block should move all the way down. To do that, 'toBottom' will be assigned `true` and 'toBottomTimer' gets a value of 0. And here, too, a sound is played.

20.3 Update

And then there's the last method, which is the center of the game: `update`. This methods sequentially performs various checks.

20.3.1 Force Down

First we check if it's time to move a block down. For this we need to increase the value of 'forceDownCounter'. If it is higher than the current game speed, the block should be moved downwards:

```
forceDownCounter += Time.d();
if(forceDownCounter > Score.getSpeed())
{
  if(canMove(currentBlock, D_DOWN))
  {
    currentBlock.move(D_DOWN);
    forceDownCounter = 0;
  }
}
```

20.3.2 Slide Counter

The slide counter is needed to still be able to move the block aside once it hits the pile. Therefore, we must first know whether the block touches the pile. In that case it can not go down any further. In this case, we will reduce the value of 'slideCounter'. When slideCounter reaches zero, we call the method **`handleBottomCollision`**.

```
if(!canMove(currentBlock, D_DOWN))
{
  slideCounter -= Time.d();
} else {
  slideCounter = SLIDE_TIME;
}

if(slideCounter <= 0)
{
  slideCounter = SLIDE_TIME;
  handleBottomCollision();
}
```

20.3.3 To Bottom

The symbol 'toBottom ' will be `true` when the player pressed the spacebar. We then move the block down quickly, but it still needs to happen gradually to be visible. So we use a counter with a small value and subtract the time delta again and again. Each time the counter reaches zero, we move the block downward. If that is not possible, the previous code (Slide Counter) will be executed.

Only when the block does not slide down, we will check the player's input.

```
if(toBottom)
{
  toBottomTimer -= Time.d();
  if(toBottomTimer < 0)
  {
    toBottomTimer = 0.05;
    if(canMove(currentBlock, D_DOWN))
    {
      currentBlock.move(D_DOWN);
    } else {
      toBottom = false;
    }
  }
} else {
  handleInput();
}
```

That concludes this method. All you need to do now is add the `create`, `update` and `draw` methods to the Game State. And then, of course, solve all the mistakes until your application works as it should.

20.4 Afterthoughts

You now have developed a complete project. Hopefully, you'll remember how important it is to turn everything into classes. The use of test applications is very important to keep a large project under control.

But of course the way we went through this project step by step does not quite match reality. The author of this course knew exactly what needed to be done and in what order you could best tackle a problem, before you even started on this exercise. If you start on your own project that will be very different. It is very common that you need to change the classes that you made. Often, methods you really need are missing because you didn't think of them before, and you will write methods which end up unused. That is, especially for a novice programmer, quite normal.

Only through experience you can learn to assess what features are really needed in a class. And even then you can not always predict them exactly.