1. HTML5 and CSS3

Design and development of web games (VJ1217), *Universitat Jaume I*

Estimated duration: 3 hours (+ 3 hours of work at home)

In this first lab session, the core aspects of HTML and CSS are introduced. As you may understand, it is not possible to cover HTML and CSS in depth in a single lab session. However, at the end of this lab you will have the essential skills to understand (and write) HTML and CSS well enough to build on. Of course, you will still need to browse books or the web to keep learning, but this will hopefully be much easier after you have worked through this introductory material. Although JavaScript will be introduced in forthcoming sessions, we will touch a bit of scripting in this session as well so that you can start getting familiar with it.

Contents

1. Introduction

World Wide Web (WWW) development —mainly meaning their services and applications— is currently centred around HTML5. HTML5 actually refers to a group of core tools employed in the development of services and applications of the so-called Web 2.0: HTML version 5, CSS, and JavaScript. As it could be expected, web games are not an exception to this rule. Current web game development is thus based on HTML5 technologies.

Roughly speaking, HTML provides the *structure* of the contents and the linkage between related items, CSS provides its *appearance*, allowing for the creation of templates which define the way all of the elements in a web page are shown, and the JavaScript code provides the *behaviour and dynamics* of a web page through client-side interaction between the page and the users.

HTML stands for *HyperText Markup Language*. Therefore, HTML is one particular kind of computer *language*: one that uses *markup* to describe the content, and has the possibility of linking the contents of different documents with *hyperlinks*.

An HTML document is essentially a text file whose contents are interpreted by the browser. We will now have a look at the structure of an HTML document through a simple example. Then, we will keep adding new elements. In our journey to learning web games development, it is important to begin with the basics and build a strong foundation to rely upon. Therefore, we do not provide you with the working code, purposefully. We want you to type it yourself so that you get acquainted with the syntax and the IDE you will be using throughout this course.

1.1. The structure of an HTML document

Let's start by having a look at a first simple example:

```
<!DOCTYPE html>
<html>
    <head>
        <title>A basic HTML document</title>
        <meta charset="UTF-8">
    </head>
    <body> <!-- This is a comment in HTML -->
        <h1>Web games with HTML</h1>
        <p>We will need to learn:</p>
        <ul>
            <li>HTML5</li>
            <li>CSS3</li>
```

```
            <li>JavaScript</li>
        </ul>
    </body>
</html>
```

Pay attention to a few simple but important things:

- We use **tags** (<html>, <head>, etc.),
- which *open* (<body>) or *close* (</body>) some contents of the document.
- You can also realise that this has a *nested*, hierarchical, structure (i.e., in other words, a tree). Thus, <h1> is inside <body>, which is in turn inside <html>.

General syntax for an HTML element

⟨element *attributes*⟩

contents

⟨/element⟩

Each *element* has an opening tag (<element>) and a closing tag (</element>). As we will see below, a few elements form an exception to this rule, and only have the opening tag.
Note the different notations between the *name* of an HTML "element" and its *tag*, <element>.
Both the *attributes* and the *contents* are optional. For some elements it is actually very common to have empty contents, since it is expected they will later on be filled programmatically.

Attributes of HTML elements

HTML attributes define visual or behavioural characteristics of the corresponding element or its contents, and they are given as a list of *(attribute, value)* pairs, as follows

attribute1=value1 attribute2=value2 ...

Although it may be tempting to separate these pairs with a comma (,), if we do this, it will not work.
The order of the attributes is unimportant. What if the same attribute is provided more than once? Although it is advisable to avoid these repetitions, in our test only its first occurrence was considered.

In our first HTML document, the <meta> tag is an example of a tag without closing tag. It has also one attribute (charset) with value "UTF-8". We will see other examples later in this session.

Your turn 1

Now, type this code, save it to a file, say, basic.html, and then open it in a browser.

- Check that the title we set with <title> appears in the title bar (it is *not* the title being displayed as part of the document itself).
- Look also at the unordered list we created with <ul>.

The actual contents of an HTML document go inside the <body>, whereas relevant information not directly related to the contents go inside <head>. The title element is a good example of this kind of *information about contents* that is not actually meant for human users to see, but for browsers, web servers and search engines. Other examples include the style, meta and script elements.

1.2. Adding style

Imagine that we now want to colour the contents of our <h1> tag. Here is one way to do it:

```
<h1 style="color: #ff00cc">
        Web games with HTML
</h1>
```

Notice that we used an **attribute** of the <h1> tag. Tags can take a number of possible attributes[1]; now we used the style attribute to set the colour. We can set many other visual **properties** of how we want things to be rendered.

Your turn 2

Add the style attribute as given above and check the resulting web page in the browser. Then, set the font size of <h1> to 30 points. **Hint:** Separate different elements of the style with ;. Let your IDE help you guess the name of the property for the font size.

Alternatively, we can set the style more generally by having this in the <head>:

```
<style type="text/css">
h1 {
 font-size: 30pt;
 color: #ff00cc; /* RGB hex values */
}
</style>
```

Please, note an important difference between the two ways: in the first case, style was a property of the h1 tag, whereas in the second case <style> is a tag *itself*. What is the advantage of the second approach? That it applies to *all* the <h1> elements of the document; we do not need to repeat the style attributes over and over for every h1 element we might have. Imagine that we had a dozen of h1 elements and after having set the style property for each of them individually, we changed our mind and decided to set it differently. Yes, it can become a nightmare to do changes this way! And the <style> tag helps us a lot. However, can we do even better? Yes, we can. Keep reading.

[1] Tip: in your future dealings with HTML5, you might want to look up an online HTML5 reference guide to see which attributes are available for each tag.

2. CSS style files

As we have seen, the HTML `<style>` tag brings a benefit when setting the appearance of the elements of an HTML document. Nevertheless, having both the contents and the description of the appearance within the same document is not the best idea. A good general principle is to keep the actual data and its representation separate. There are many reasons for that, a simple one being **modularity**: the fact that we can change one thing (the contents) or the other (its appearance) *independently* much more easily. **Reusability** is another very good reason: we can design the appearance *once* and reuse it *multiple* times, for different documents. This way, not only the effort pays off, but we can gain **consistency**: a set of related documents sharing a common style definition will have the same *look & feel*, which is a must for professional content delivery.

That's fine, but how can we achieve this contents-presentation separation in practice? Simple: by using CSS (*Cascading Style Sheets*) files.

Therefore, we are going to remove the `<style>` tag from our HTML file and write its contents to a separate file.

Your turn 3

Create a folder `css` at the same level where you have the `basic.html` file. Create the file `basic.css` inside this new `css` folder, and type in there the contents of our `<style>...</style>` (only the contents, not the tags!). Don't forget to remove all the styles in the `basic.html` file.

As you may guess, this is not enough to apply the properties described in `basic.css` to the contents of our `basic.html`. In addition to this, we need to *relate* them. To that end, we need the following code snippet inside the head section of `basic.html`

```
<link rel="stylesheet"
 type="text/css"
 href="css/basic.css">
```

Here we have a new tag, `<link>`, whose purpose is to *load* some contents which are in some other file elsewhere. Notice the three new attributes and their values:

- `rel="stylesheet"` indicates how the linked file *relates* to the HTML document (mandatory). Here, we say we are referring to a style sheet;
- `type="text/css"` indicates the *type* or format of the file (optional). Here we indicate that it is a text file following the CSS syntax;
- `href="css/basic.css"` indicates which is the file and where it is. Here we say the file is `basic.css`, and it is inside a folder we have named `css`. (Box 1)

You can think of `<link>` as the equivalent of directives in other languages: `import` in Python, `#include` in C/C++, etc. See Box 2 for uses of `<link>` other than loading CSS files, and Box 3 for a good summary of three ways to use CSS.

Box 1. Use relative paths

When specifying file locations (as in the `href` attribute), paths relative to the location of the main HTML file are often used. Typical examples of paths might thus be `css/style1.css`, and `js/main/utils.js`. If we needed to access some file higher in the file hierarchy, we might use `..` (remember this means *"go up one level"*). For instance, `../../libs/js/utils.js` goes up two levels and then goes down two levels along folders `libs` and `js`, where the file `utils.js` is expected to be.

Box 2. Linking other resources

Besides loading CSS style files, the `<link>` tag can be used to load other resources such as JavaScript libraries, or font files. These files can either reside locally (at the server) or remotely.
For instance, the so-called **web fonts** are typically loaded from some external resource. You can visit the Google Font Directory, http://fonts.google.com, and find a font you like. After that, you can get the font embedding code. For instance, if you wanted to use Gochi Hand, you would include either

```
<link href="https://fonts.googleapis.com/
 css?family=Gochi+Hand" rel="stylesheet">
```

or

```
<style>
@import url('https://fonts.googleapis.com
        /css?family=Gochi+Hand');
</style>
```

in your HTML document, and you would include this:

```
font-family: 'Gochi Hand', cursive;
```

as a declaration in some rule of your CSS file.
You can choose appropriate distinctive fonts to your own games. Remember that the fonts used convey messages, intended or not. Therefore, if used well, they are powerful graphical elements to exploit.

CSS files consists of a series of **rules** whose syntax we have already seen. Its general form is as follows:

A CSS rule looks like this:

selector { property: value; }

This reads: "set the given *value* to the given *property* for the HTML elements given by *selector*".
The part "property:value" is called *declaration*. There can be several declarations within the same rule, separated by ; The order of the declarations is unimportant. What if the same property is repeated? In our test, only the last value was considered. It is though preferable to avoid repeating the same property within the same rule.

There are many rich and flexible ways to indicate the selector, but in its simplest form, a selector can be:

1. all HTML elements (*), to apply a common style to all elements;
2. all HTML element of a certain type (e.g., all list items `<li>`);
3. a selection of HTML elements, denoted by a class;
4. a particular HTML element, denoted by an identifier.

So far, we have only seen the second possibility (all HTML element of a certain type), when we set the colour and font size of (all) the `<h1>` tag(s). Now we introduce how to select elements more fine-grained, using *classes* and *identifiers*.

3. Identifiers and classes

Imagine we have three `<h1>` elements in the document, and we want to apply a particular style to just *one* of them. With the rule defined above, this would not be possible, since *all* `<h1>` are affected equally by the rule. What can we do then? Well, we can use the `id` attribute for the particular `<h1>` element we want to style and then refer to it in the CSS rule.

Your turn 4

Before we see how to use the `id` attribute, add a new `<h1>` element, `<h1>Useful books</h1>`, for instance, to `basic.html`. Check that the CSS rule applies to this new `<h1>` as well.

Now, pretend we want to apply the colour and font size *only* to this new `<h1>` element.

Your turn 5

1. Add the `id` attribute:

```
<h1 id="books">Useful books</h1>
<!-- Notice the new attribute, "id" -->
```

2. Replace the selector `h1` of our CSS rule by `#books`. The rule then becomes:

```
#books {
 font-size: 30pt;
 color: #ff00cc;
}
```

Pay attention to the syntax: we have a # before the identifier. This is different to when we use the name of an HTML element.
Verify the result is as intended.

Identifiers are meant to be unique. So, do not use the same `id` attribute in more than a single HTML element. Actually, when we want to apply the same style to a selected set of HTML elements, we use classes. Let's see how.

```
<h1 class="resources">Web Links</h1>
<h1 class="resources books">Useful Books</h1>
```

Notice that the first `<h1>` has been assigned to class `resources`, and the second `<h1>` has been assigned two classes: `resources` and `books`. You may rightfully be wondering what can this be useful to. Look at this new CSS file `basic.css`:

```
.resources {
  font-variant: small-caps;
}
.books {
  color: blue;
}
```

Notice that the syntax to refer to a class in CSS is `.class-name` (i.e. with a *dot* before the class name).

Your turn 6

Update your HTML and CSS files to reflect these changes. Try to predict their effect. Only after that, try it on your web browser. Make sure you understand what happened and, more importantly, *why*.

Notice we set one colour for the `.books`. What would happen if we set *another* colour, like this?

```
<h1 class="resources books" style="color:pink">
Useful Books
</h1>
```

Your turn 7

Will this `<h1>` element be rendered in blue or in pink? Any guess why? Keep reading.

This is an example where different rules may contradict one another. In these cases, the general rule is very simple: *the last style defined is applied*. This explains why the colour defined inline here dictates the final rendering colour. Dynamic changes are also possible (Box 4).

Styles can be defined depending on the *context*. For instance: imagine we wanted elements of class `book` to be boldfaced only if inside `<h1>`, we might have these CSS rules

```
.books { /* class "books" no matter where */
 color: pink
}
h1.books { /* contextual rule: only applies to
         class "books" in h1 elements */
 font-weight: bold;
}

/* Be careful: The following is *not*
 an example of contextual rule */
p, h1, h3 { color: red; }
/* It applies the same style to different
elements,  separated by , (not by space) */

 /* Using space between selectors would denote
 a hierarchical relation (parent - child) */
```

Although more complex contexts can be defined, an advanced treatment of this topic is out of the scope of this course.

Box 3. Three ways to use CSS

Remember that we have seen three ways to use CSS:

Inline As an attribute of a particular HTML tag:

```
<h2 style="color:red; background:green;"> This is a red text with a green background </h2>
```

Embedding As an `<style>` tag within the HTML document:

```
<style type="text/css">
 h2 {
 color:red;
 background:green;
 }
</style>
```

Linking By indicating which CSS file contains the style rules:

```
<link rel="stylesheet" type="text/css" href="mystyles.css">
```

We can include a CSS file from within another: `@import "newstyles.css";`

Box 4. More on classes

The membership of HTML elements to classes can be changed programmatically. As a consequence, style properties can be varied dynamically by adding or removing classes. We will see how later in our course.

4. Some more HTML elements

Grouping block elements

The `<div>` tag is both very general and widely used. It defines a division in an HTML document. It can be empty so that its contents can be defined programmatically, but it usually contains other HTML elements so that they can be styled jointly with CSS. This is possible because, as with any other HTML tag, `<div>` can also have the `style` attribute and, of course, the `id` and `class` ones.

See how

The `<div>` tag

Grouping inline elements

The `<span>` tag is the *inline* counterpart to `<div>`, and it therefore groups inline elements that could be styled together.

See how

The `<span>` tag

Box 5 briefly discusses block and inline elements.

Box 5. Block and inline elements

HTML elements have a default display value, typically *block* or *inline*. They differ in that a block-level element starts at a new line and takes up the full width of the document, whereas an inline element does not. Some examples of each are:

- Block-level:

`<div>`	`<p>`
`<h1>`–`<h6>`	`<li>`
`<table>`	`<canvas>`

- Inline:

`<span>`	`<a>`
`<style>`	`<script>`
`<img>`	`<button>`

Tables

Contents in a web page can be organised in tabular forms with HTML tables.

See how

HTML tables

It is good to pay attention to how easily we can style differently the header cells (`<th>`) and the regular cells (`<td>`), or add a border.

Images

Images can be included with the `<img>` tag. This tag has neither contents nor a closing tag. The image to be loaded and some attributes such as its size are given with their attributes.

See how

HTML images

Like `<img>`, a few more elements have no closing tag, such as the simple horizontal rule (`<hr>`), and two we are already acquainted with: `<meta>` and `<link>`. These elements can be referred to as *empty* elements and sometimes also as *singleton* or *void* elements.

Hyperlinks

The `<a>` tag defines hyperlinks, and is therefore one of the key ingredients of the web.

See how to use it

HTML `<a>` Tag

See Box 6 for a reminder of what a hyperlink is and some notation clarification.

As you know, the links are conventionally rendered as underlined blue text if not visited, or as underlined purple if visited. Often, a link also changes appearance if we hover over it, or while we are clicking it. CSS styles can be applied to modify these "default" or common appearances, or in more technical terms, we can manipulate the appearance of a link depending on *the state* it is in (i.e., not visited, visited, hovered, active).

CSS pseudo classes

Just as we can define classes to denote a selection of HTML elements, CSS also foresees some pre-defined classes, called *pseudo classes*. They denote a certain state an element is in, and are selected using a colon:

selector : pseudo-class

All available CSS pseudo classes

See how to style different states of a link

Styling links

Your turn 8

Add a link to one of your favourite web pages and style it so that unvisited links have a yellowish background, which turns bluish once visited. If you want, apply also an style you find appropriate when the link is hovered over.

UI widgets

User Interface (UI) widgets are necessary components for interacting with the user. In web games, they can be useful to set some aspects of the game such as whether sound is on or off, among many others.

Have a look to buttons, for instance

HTML `<button>` tag

If you followed the Try it Yourself, you will have noticed an important attribute: `onclick`. This attribute, and others similar to this one, allow us do define some "behaviour" in response to some "event". In this case, upon the user clicking on the button (the event), an alert box pops out (the behaviour). Certainly, this is a simplistic example, but you can get the idea of how this works and what this can be useful for. This relates to the paradigm of *event-driven programming*, which is at the core of interactive applications with a graphical user interface, which is the case not only of desktop and mobile applications but also of web browsers. This concept is very important and we will come back to it over and over again.

Scripts

Just like we can have CSS style embedded in the HTML document, so we can with JavaScript (JS) code. For long programs, it is advisable to have this code in separate files. For short scripts, though, we can just include the code inside the `<script>` tag.

Look at a simple example

HTML `<script>` Tag

In particular, this example illustrates how we can access the element with identifier `demo` within the HTML document (`document`). Notice also that we can set or change the contents of this HTML element by setting `innerHTML`. As we will see in a moment (Sect. 4.1), we can also read and write CSS properties.

4.1. An example

For practising what we have learned in this section, let a checkbox control the background colour of a `<div>` element. The overall description of what we need is:

- Insert the `<div>` with an appropriate identifier so that we can later refer to it.
- Insert a checkbox and have its `onclick` property indicate which code to run upon user click. Again, we need an identifier for the checkbox so that we can consult whether it is checked.
- Write the code that checks whether the checkbox is checked and set the background colour of the `<div>` accordingly.

Your turn 9

Follow the steps below. Just after each step and before proceeding to the next one: (1) try its effects on the web browser; and (2) make sure you understand what is happening and why we wrote the corresponding code.

1. Define one `<div>` with `id="background"` that surrounds two `<h1>` elements.
2. Include a checkbox, as follows:

Box 6. What is a hyperlink?

The notation about links can sometimes be confusing. Let's try to clarify it.

Term	Meaning
Link or hyperlink	Data that can be followed by clicking (or tapping or hovering)
Anchor text	The visible, clickable text in a link which identifies what is being linked
Hyperlink reference	The target URL

We may therefore formulate:

$$\text{Hyperlink} = \text{anchor} + \text{reference}$$

Thus, if we look again at a hyperlink in HTML, we can not only easily spot the two parts (anchor and reference)

```
<a href="http://www.w3.org">Anchor text</a>
```

but also understand what is behind the names of the tag `<a>` (after anchor) and the attribute `href` (after hyperlink reference).
Alternative names for *anchor text* are *link label*, *link text*, and *link title*.
Although the *anchor* is usually text, it can also be any other HTML element, such as an image.

```
<input type="checkbox"
 id="back-gray"
 value="gray">
Gray background
<br>
```

3. Add the attribute `onclick` to the checkbox with value `"setGray()"`
4. Include the following JS function inside the `<script>` tag:

```
function setGray()
{
 let color="white"; // value if not checked
 if (document.getElementById("back-gray").checked)
   color="gray"; // value if checked

 // get element with desired id
 const e = document.getElementById("background");

 // set the background color to that element
 e.style["background-color"] = color;

}
```

Don't worry if you do not understand every detail of this code. For now it is more important that you really get the general gist of what is being performed. We will go back to some of these things, in particular to JS programming, and how to work effectively with editing and viewing HTML (Box 7 for a preview).

Box 7. Opening HTML files

When developing web applications, we'll be creating and updating HTML/CSS files locally, and we'll frequently want to see the result to verify if we are on the right track. Of course, we can use our favourite file explorer, browse to the relevant directory and open the local file using our favourite browser.
This works just fine for basic HTML/CSS (and JavaScript). But, it's a bit tedious to continuously switch from our development environment (Visual Studio Code) to our file browser, and furthermore, once we'll get into more advanced HTML programming, we'll get into trouble (mainly due to additional security restrictions applying to local files). And of course, when using server-side HTML programming —which is beyond the scope of this course— our pages won't render.
Therefore, it is good practice to always open local HTML files using a local web server, as to mimic the environment they'll eventually run in (i.e., an online web server). There are many (free) web servers we can download and install, but luckily, Visual Studio Code provides an extension that does it all for us: launch a local web server, open the file in this web server, and keep a live connection ensuring the rendered HTML file updates as we change it. This extension is called *LiveServer*; please refer to **Visual Studio Code usage tips** at Appendix A.

5. More CSS properties

5.1. Beyond font properties

It is important to realise that styling HTML elements can actually go beyond simple colour fonts or colour sizes. For instance, boring lists of links can be turned into good-looking navigation bars.

From boring lists to cool navigation bars

CSS Navigation Bar

Going even further than this, the structure of the HTML documents can also be defined via CSS properties.

See how

Website Layout

Eventually, the appearance of a whole HTML document can be drastically changed depending on the style sheet being loaded.

Try this illustrative example

One HTML Page - Multiple Styles!

Finally, just for fun, have a look at this world-wide challenge to show off the power of CSS.

The beauty of CSS Design

CSS Zen Garden

5.2. The value of templates

If you try the examples above you will realise the importance of CSS templates to format an HTML document. Different styles or templates can be applied without changing the contents of an HTML file. You only have to design your style for a given document, write its rules to an external CSS file and load this file in the HTML document. Moreover this same style can be applied to other documents, thus giving all of them the same look and feel.

Your turn 10

- Create a new folder to store the files you'll use in this exercise. Let's call it *Exercises* (you can give it another name if you prefer).
- Download the archive `testingStyles.zip` from *Aula Virtual*, copy it to the *Exercises* folder and unzip it.
- Open *Visual Studio Code*, and open the *Exercises* folder using File - Open...
- In Visual Studio Code, browse to the `index.html` file and open it. You now see the source code of this file (i.e., the HTML elements).
- Let's check out how the page looks like! (see box *Opening HTML files*)
- Now, apply to it the two CSS files in the ZIP archive —one after the other, obviously, and check out what it looks like. Be aware that one of these styles requires an image to be loaded —check the path!

5.3. Animating elements

With CSS we can set things other than static properties. With an animation, the style of an element is changed *gradually*.

Look at examples

CSS Animations

As you can see, to define an animation, we need to specify:

- Which element(s) we want to apply the animation to. This can be achieved by setting an animation name.
- Which properties we want to change. They can be the colour, the position, the size, etc.
- The duration of the animation and, optionally, a delay to start it after the page is loaded.
- The values of the properties at some keyframes. They can be expressed relatively as percentages of the total duration (e.g. "the colour at 30% of the animation should be green"), or simply by using the keywords `from` (0%) and `to` (100%).

By the way, notice how we can pretend that we have a rectangle drawn just by setting the width, the height and some background colour of a `div` element.

Your turn 11

Insert an image (remember: use the `<img>` tag) and let it translate from left to right while rotating clockwise up to 90° and then, at half-time of the animation, translate it from right to left while rotating it anticlockwise.

Hints:

- Set the CSS property `position` to either `absolute` or `relative`.
- Use the `transform` property. For instance, the specification for the keyframe at 50% will be like:

```
50% {left:80%; transform:rotate(90deg);}
```

6. The *canvas* element

One of the most idiosyncratic elements offered by HTML5 which was not available before is the **canvas**. On the canvas we can draw primitives (rectangles, circles, polygons, etc.), and place text and images. By clearing its contents and drawing new contents again at a fast rate, the illusion of movement can be created. Since many or most HTML5 games rely on the canvas, you should place particular attention to it.

A typical inclusion of a canvas element will be like this:

```
<canvas id="myCanvas"
 width="200" height="100"
 style="border:1px solid #000000;">
Your browser does not support the HTML5 canvas tag.
</canvas>
```

Notice that we set the canvas' width and height and, very importantly, specify an `id` attribute. As with any other HTML element, this will allow us to access it from within JavaScript code. It is customary to write a text inside the canvas element which would be displayed in those browsers which don't implement this element.

Since we have already studied how to include some simple code snippets in JavaScript, you are in a good position to draw some simple graphics.

Look some examples first

- HTML5 Canvas

You can have a look at Box 8 for a reference of some drawing functions.

Your turn 12

1. Define a canvas element in your HTML file. Don't forget to specify an identifier.
2. Using the examples as a guide, draw a filled circle of pink colour and transparency factor of 0.3. The transparency ranges from 0 (fully transparent) to 1 (opaque).
   - Place the circle at the centre of the canvas.

     Hint: If `c` is the JS variable referring to the canvas element, then `c.width` is the canvas' width.
   - Let its radius be 40% of the smallest between the canvas' width and height.

     Hint: Use `Math.min(a,b);`

   Remember that the angular parameters to the `arc()` function should be in radians, and that π radians $= 180°$. You can use the predefined constant `Math.PI` for π.

Could we easily detect the user clicking inside the drawn circle? Well, it is not difficult, but it is not straightforward either. The most important thing to bear in mind is that once we draw something on the canvas, we lose its information of where it is. If we need to know where the circle was drawn and which size it has, we need to keep this information somehow. Then, when the user clicks on the canvas, we should work out ourselves whether the mouse cursor position is inside the circle or not. Checking for collisions between game entities, each with its own geometry, is also regularly required in many games. Fortunately, some game frameworks (such as Phaser, which will be studied in this course) perform these kinds of calculations for us, which can facilitate our programming tasks.

Your turn 13

Let a widget control some aspect of your drawing. For instance, the opacity of the circle can be set via a slider.

1. Create the slider in the HTML document.

   Hint: Use `<input type="range">` with the proper attributes
2. Add the attribute `onchange` to the slider, and set it to a JS function, `setOpacity()`, for instance.
3. Add the corresponding code to read the slider value and use it to set the opacity of the circle.

   Hint: Complete the following code snippet:

```
let slider;
setOpacity();
function setOpacity()
{
 slider = ... // read the slider's value
 redraw();
}
function redraw()
{
 // Essentially the same code we had before,
 // but now using the variable slider
 // to set the opacity ("alpha") parameter
}
```

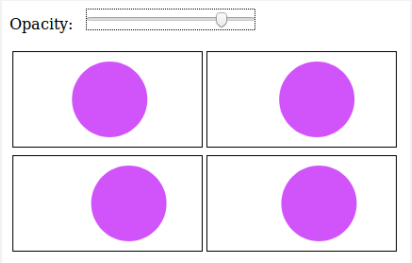
   If you need to concatenate strings, the operator + does the expected job.

Can we have more than one canvas inside the same HTML document? Yes, indeed. What can this be useful for? Well, it is not a very common situation, but we should strive to be creative (Box 10) We might consider to have separate games (replicas of the same one or just different ones) in different canvases or, similarly to TWITCH, the game can be distributed among the different canvases. What else can you imagine?

Your turn 14

As a final task, create a table where the contents of each cell is a canvas. It can be the canvas with a circle we have been working on. Although initially drawn centred on each canvas, let the circle move or change its size randomly a bit each time the users click on the (common) slider to change their opacity.

Here is a snapshot of our 2×2 table of canvases:



Keep exploring some more possibilities on your own.

It is good and fun to know that we can draw some shapes without using the canvas (Box 9).

References

To prepare this document we have found inspiration in, basically, these books:

- Makzan. *HTML5 Games Development by Example*, Packt Publishing, 2011
- Jason Cranford Teague. *CSS3. Visual QuickStart Guide*, PeachPit Press, 2011
- John David Dionisio and Ray Toal. *Programming with JavaScript*, Jones & Barlett Learning, 2013

Box 8. Drawing on a canvas

Here are some JS functions for drawing some primitives on a canvas:

Function	Purpose
`fillRect(x, y, width, height)`	Draws a filled rectangle
`strokeRect(x, y, width, height)`	Draws a rectangular outline
`clearRect(x, y, width, height)`	Clears the specified rectangular area and makes it fully transparent
`beginPath()`	Starts recording a new shape
`closePath()`	Closes the path by drawing a line from the current drawing point to the starting point
`fill()` `stroke()`	Fills or draws an outline of the recorded shape
`moveTo(x, y)`	Moves the drawing point to (x, y)
`lineTo(x, y)`	Draws a line from the current drawing point to (x, y)
`arc(x, y, radius, startAngle, endAngle, anticlockwise)`	Draws an arc at (x, y) with specified radius; we set anticlockwise to false for clockwise direction
`strokeText(text,x,y)`	Draws an outline of the text at (x, y)
`fillText(text,x,y)`	Fills out the text at (x, y)
`fillStyle()`	Sets the default colour for all future fill operations
`strokeStyle()`	Sets the default colour for all future stroke operations
`drawImage(image, x, y)`	Draws the image on the canvas at (x, y)
`drawImage(image, x, y, width, height)`	Scales the image to the specified width and height and then draws it at (x, y)
`drawImage(image, sourceX, sourceY, sourceWidth, sourceHeight, x, y, width, height)`	Clips a rectangle from the image (sourceX, sourceY, sourceWidth, sourceHeight), scales it to the specified width and height, and draws it on the canvas at (x, y)

Box 9. Card games and drawing shapes with CSS

It is possible to easily build some games without using the canvas at all. In particular, when game elements are rectangles, they can readily be displayed as `<div>` elements. This is the case of card games such as matching cards (memory game). As an example:
Play one of such games
Browse or study its code
Actually, we can draw primitives other than rectangles with a few tricks. For instance:

- with *rounded* corners, we can make a square look like a circle! (Example)
- with a zero-sized `<div>` and a *border* of some thickness, we get a triangle! (Example)

Look at more amazing shapes

Box 10. Think outside the box!

In many professions, and (web) game design is just one of them, being truly creative is of uppermost importance. Makzan, the author of the book *"HTML5 Games Development by Example"*, mentions some examples of web games that are unconventional.

- With URL hunter! the game takes place on the URL bar. Have a look. It is truly original.
- TWITCH is a series of connected mini games where each minigame takes places on a small browser window.

Makzan says: "With creativity, we are not bound in a rectangle game stage any more. We can have fun with all the page elements [...]"

A good place to know HTML elements or find CSS properties is `https://www.w3schools.com`:

- HTML elements
- CSS properties.

7. Exercises

1. If you feel like, you can informally self-assess your knowledge of HTML and CSS with these quizzes:
   - HTML Quiz (40 questions)
   - CSS Quiz (25 questions)
2. Let's use CSS to draw a toggle switch. First, we'll focus on the visual aspect, and create a static on and off switch (i.e., no behaviour yet: they cannot be flipped). Have a look at our toggle switches, and try to mimic them as closely as possible:

We include one of such toggles like this:

```
<div class="toggleSwitch">Sound
 <div class="switchOn">on</div>
</div>
```

Tricky. Remember, CSS supports rounded corners by default, and using some tricks, we can create circles as well (Box 9). The rest is a matter of positioning, and lots of trial and error.

There are several ways to achieve the solution, but probably, the following properties will come in handy: positions, margins, borders and padding. Hint: some of these properties accept negative values.

Then, make your switch look nicer, perhaps using some shadows and text styling.

3. Repeat the toggle exercise, but this time drawing it on a canvas instead. We got these less cute toggles:

Can you do better?

We include one of such toggles like this:

```
<canvas id="Sound" width="120" height="35"
        style="display:block">
</canvas>
```

Again, no behaviour needed just yet, just draw the toggle switches.

4. Now, let's make our toggle switches functional, and add the behaviour of flipping them. We start with the CSS version. To do so, we call a JavaScript function `switchToggle(this)` when the toggle is clicked, which will dynamically change the class of the current element (which will immediately change its appearance), and switch its content (from *on* to *off*, and vice versa). Also, don't forget a nice transition (using CSS).

   Our HTML element including the toggle switch now looks as follows:

```
<div class="toggleSwitch">Sound
 <div onclick="switchToggle(this)"
           class="switchOn">on</div>
</div>
```

   You might want to wait to solve this exercise until after lab 2, where we'll see basic JavaScript and how to programmatically interact with HTML elements. But hey, if you feel brave, dive right in, and write the `switchToggle` function!

5. Finally, let's add the same behaviour to the canvas version of our toggle switch. To do so, we'll again add the `onclick` attribute, this time to the `canvas`, and call a function which will achieve the toggle flip. However, the way to achieve this is totally different: as we completely manage the canvas ourselves, the function will now need to (re)draw the correct (flipped) toggle switch.

   Our solution will also need some more management: keeping track of the different canvases, their state, etc. For example, we could foresee the following datastructures (but note: there are different options):

```
const toggle_id = ["Sound", "Music"];
const toggle_state =
{Sound: true, Music: false};
```

   Again, only for the brave! Next week, we'll see the necessary tools to solve this exercise, so feel free to revisit this exercise after next week's lab.

A. Visual Studio Code usage tips

Built-in support Visual Studio Code (VS Code) is a very powerful, versatile multi-language editor. Despite it being multi-language, it still provides quite a lot of built-in support for each specific programming language, including HTML, CSS and JavaScript. Think of syntax highlighting, bracket matching, code formatting, code folding, code peeking, code completion, parameter info, documentation info (the latter three are part of what Microsoft calls IntelliSense), and many other useful features.

Extensions The real power of VS Code comes with *extensions*. There are tons of extensions available, each providing specific additional support, some generally adding features to the editor, but most providing dedicated support for a particular programming language. In the *Aula Virtual*, there is a list of extensions available we consider useful —please check it out— but we also encourage you to browse through the available extensions and install those that appear useful to you.

Learn about the VS Code extension marketplace.

The Command Palette Most of Visual Studio Code's functionality is available through the *Command Palette*, a context-aware auto-completed command line tool that is accessed through the keyboard shortcut *control-shift-P* (Windows) or *command-shift-P* (Mac). You'll also find the keyboard shortcut of each command there, which you can gradually memorize to increase your coding productivity. Browsing through the command in the Command Palette is an easy way to get a quick idea of all the functionality you have available.

IntelliSense We have mentioned IntelliSense before, but given its importance, we'll go into a little bit more detail here. IntelliSense is sometimes dubbed as context-aware autocompletion, but in fact it's a set of tools (including:

code completion, parameter info, quick info, member lists) that can enormously improve your productivity, and reduce errors. Using it is pretty straightforward: as you type, IntelliSense will try to guess what you want to type, and provide you with suggestions. To insert a suggested item in your code, just select it and press tab or enter.

Have a look at VS Code IntelliSense.

Get to know Visual Studio Code Knowing what the editor, and its extensions, can do for you, is essential to program efficiently and productively. We highly recommend you to invest some time now to study the documentation and go through some tutorials, in order to save a lot of time later on. To help you, we made a selection of useful resources, with the boldfaced links being **highly recommended**.

The following links are for text-based documentation:

- **VS Code official documentation**
- **10 productivity tips for VS Code**

And these are video tutorials:

- Installation, basic settings, configuration
- **Useful shortcuts and functionality**
- A longer introduction for beginners

There are many other online resources available, some much more extensive than the above. Use you favourite search engine if you are interested in finding more.