{EPITECH}

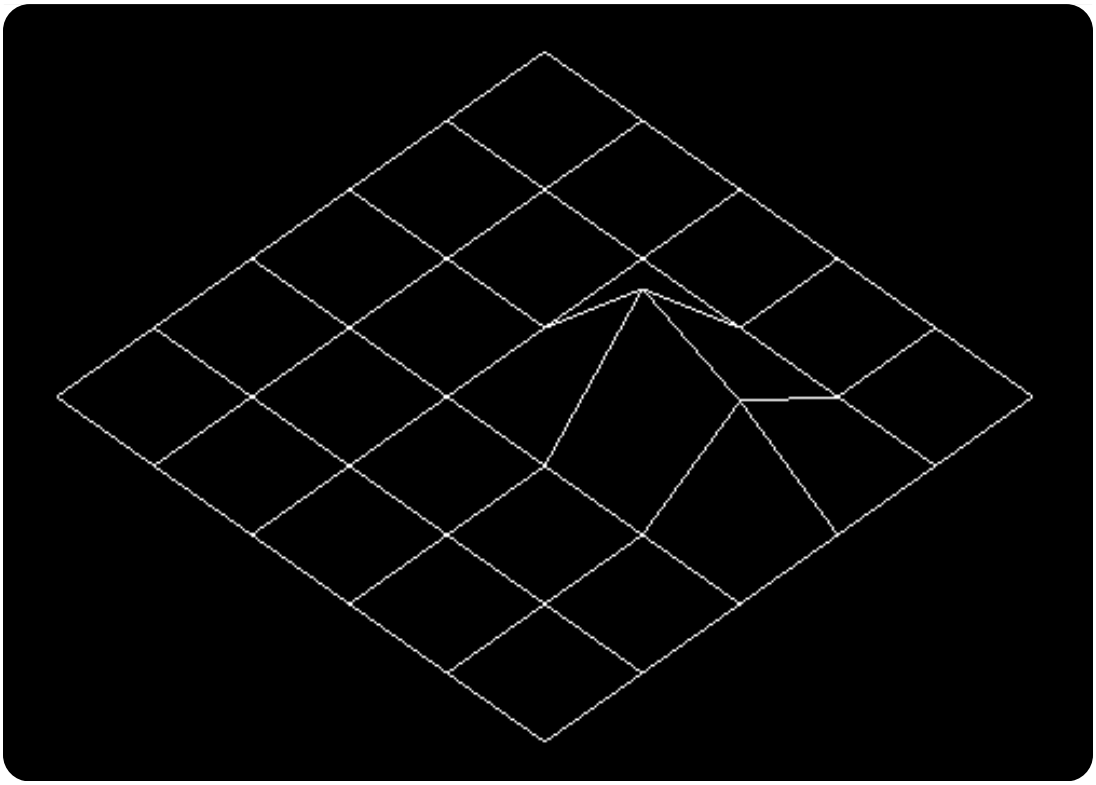
BOOTSTRAP MY_WORLD

INTRODUCTION TO 3D PROJECTION

BOOTSTRAP MY_WORLD

There are several interesting ways to simulate a 3-dimensional space: parallax scrolling, objects shadowing, objects partially hidden by others or perspective projection. In this bootstrap, we will go through a technique called **isometric projection**.

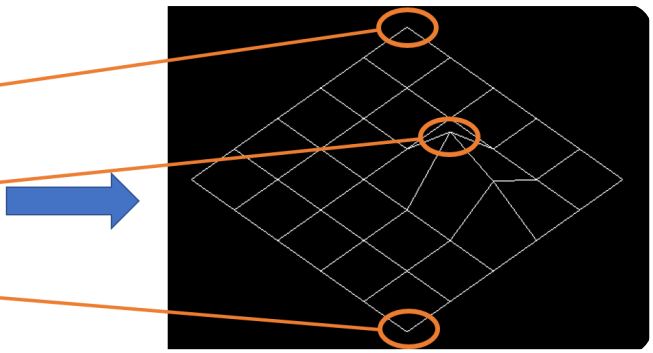
This technique consists in displaying a 3-dimensional map in a 2-dimensional window from a particular viewpoint, as in the following image:



Transform a 3D-map into a 2D-map

3-dimensional maps can be represented as `int **`. The indexes are both the x and y coordinates of the points of the map. The values within the map are the altitudes of these points:

```
#define MAP_X   6
#define MAP_Y   6

int map[MAP_Y][MAP_X] = {
  {00, 00, 00, 00, 00, 00},
  {00, 00, 00, 00, 00, 00},
  {00, 00, 00, 05, 03, 00},
  {00, 00, 00, 00, 00, 00},
  {00, 00, 00, 00, 00, 00},
  {00, 00, 00, 00, 00, 00}
};
```



First, create a function `project_iso_point` that transforms a 3-dimensional point into a 2-dimensional one. It should be prototyped as follows:

```
sfVector2f project_iso_point(int x, int y, int z);
```

The isometric projection could be calculated as follows:

```
2d_point.x = cos(angle_x) * 3d_point.x - cos(angle_x) * 3d_point.y;
2d_point.y = sin(angle_y) * 3d_point.y + sin(angle_y) * 3d_point.x - 3d_point.z;
```

You can try with *angle_x* equal to 45° and *angle_y* equal to 35°, but they must be defined in radians.

It's OK if these formulas seem like magic incantation. But with a minimum of curiosity, research and serendipity, you should be able to understand them quite easily.

Write another function, named `create2dMap`, that transforms a whole 3-dimensional map into a 2-dimensional one. It simply loops over all the points of the map and store their isometric projection coordinates. It should be prototyped as follows.

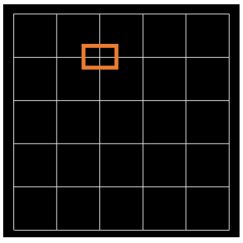
```
sfVector2f **create_2d_map(int **3d_map);
```

(You must use your `project_iso_point` function that you just created.)

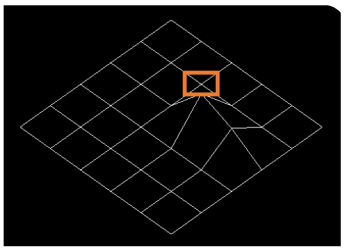
In this function, you need to consider the sampling that must be performed to transform the coordinate in the map into the coordinate within the window. For example, here there is a point every 64px along the x and y axis.

```
#define MAP_X   6
#define MAP_Y   6

int map[MAP_Y][MAP_X] = {
  {00, 00, 00, 00, 00, 00},
  {00, 00, 00, 00, 00, 00},
  {00, 00, 00, 05, 03, 00},
  {00, 00, 00, 00, 00, 00},
  {00, 00, 00, 00, 00, 00},
  {00, 00, 00, 00, 00, 00}
};
```

(2 ; 1) in the map



(128px ; 64px) in the window



(128px ; 64px) in the window (iso)

(Place the sampling in a macro, and modify it to witness the changes it generates.)

Display a 2d wireframe

Let's start from the `sfVector2f **` previously created with the `create2dMap` function. In CSFML, there is a simple way to draw a line from the two points delimiting the line. To do so, you can use sfVertexArrays since they are drawable by your `sfRenderWindow`.#br

The following function creates and returns a drawable sfVertexArray from two sfVector2f.

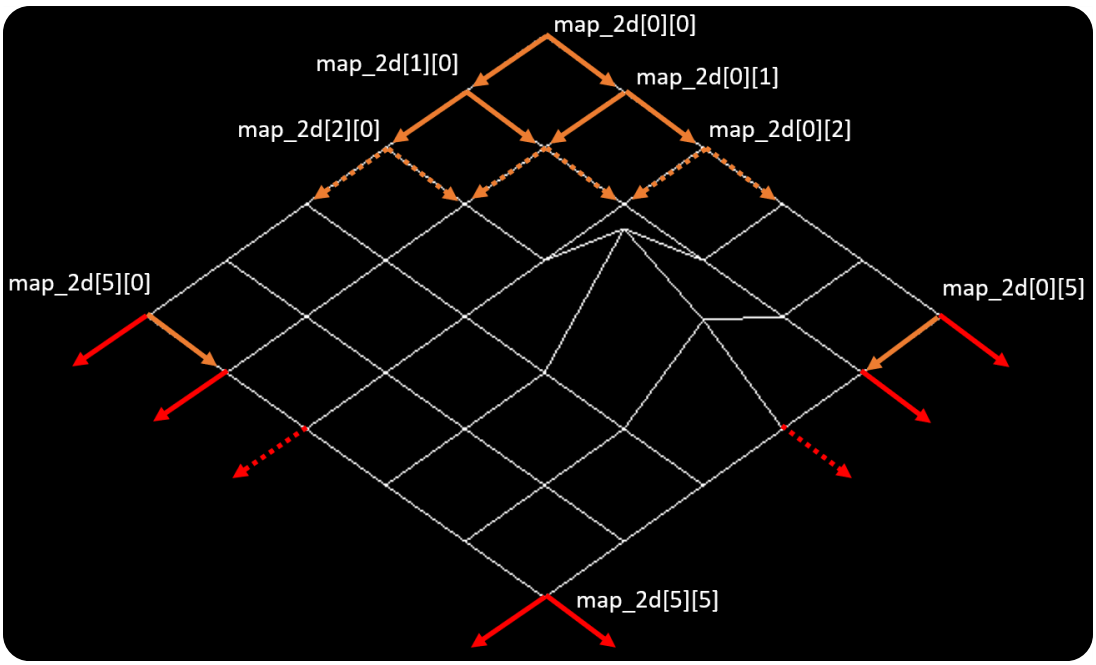
```
sfVertexArray   *create_line(sfVector2f *point1, sfVector2f *point2)
{
        sfVertexArray *vertex_array = sfVertexArray_create();
        sfVertex      vertex1 = {.position = *point1, .color = sfWhite};
        sfVertex      vertex2 = {.position = *point2, .color = sfWhite};

        sfVertexArray_append(vertex_array, vertex1);
        sfVertexArray_append(vertex_array, vertex2);
        sfVertexArray_setPrimitiveType(vertex_array, sfLinesStrip);
        return (vertex_array);
}
```

Now, write the `draw_2d_map` function that takes a 2-dimensional map as parameter and makes it drawable in the window on the next call to `sfRenderWindow_display`. It should be prototyped as follows:

```
int draw_2d_map(sfRenderWindow *window, sfVector2f **2d_map);
```

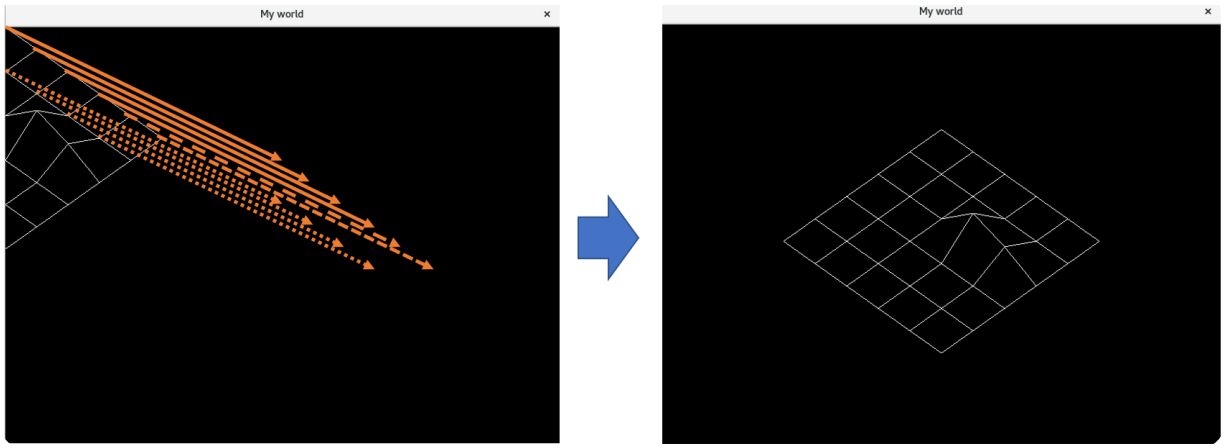
This function loops over all the points and links them to their direct neighbour.

(Mind the points on the edge of the map.)



Center the displayed map

If your map is not centered in the window, you only need to apply a translation (on both x and y axis) to every point after they have been projected.



v2