Lecture 11

Binary Collision Hot Spots

1. Platformer – Binary Collision Map 2
    1.1. Introduction 2
    1.2. Initialization 3
    1.3. Point collision 4
    1.4. Sprite collision using hot spots 4
    1.5. Snapping 7
    1.6. Normalized coordinates system 8
    1.7. Flipping the Y value 11
    1.8. Recapitulation 13

CSD1130
Game
Implementation
Techniques

1. Platformer – Binary Collision Map

1.1. Introduction

- In games, players usually do not have a complete freedom while controlling the main character (or any controllable sprites). The movement is dictated by game play rules:
    - In platform games, the main character can move horizontally on platforms, with the ability to jump a small distance vertically.
    - In a “bricks” style game, the pad can only move horizontally.
    - In maze-based games, where each level is divided into 2 areas, usable and unusable, game objects can only maneuver in “usable” areas.
    - The same logic applies in 3D games, the player has always some kind of restrictions concerning moving and controlling game objects.

- Example: Platform games.
    - Movement of the main character is restricted: It can only walk on platforms.
    - Each level is divided into 2 main parts: Platforms and non-platforms (air).
    - World collision is “static”: Sprites can collide with the left side of a wall when moving right, or with the bottom side of a wall when moving upwards etc.

- When the main character moves in any direction, we must check for collision against the platform walls.
    - We can apply the traditional collision check, which uses the sprite's collision data.
    - Usually, a bounding circle or rectangle.
    - This will force us to create a bounding rectangle on each platform and wall in each level for the collision check to work.
    - This requires a lot of memory to store the collision data of each platform wall, it will be expensive computation wise since it will involve lots of collision checks between game objects and all the potential walls (according to its movement direction).
    - Also, simple collision functions just check for collision, they do not provide the collision sides.

- Instead of applying a traditional collision check for this case, we can use a binary map.
    - It consists of dividing the level into a grid.
    - Each cell inside this grid can hold 1 of 2 values: 0 or 1.
    - If the value of a certain cell is “0”, then it is considered as a non-collision area.
    - On the other hand, if its value is “1”, then it is considered as a collision area.

- Binary map collision can be used for other game types like Pacman.
    - Similarly, to how it is used in platform games, we can create a grid where the "0" cells represent the areas where Pacman and its enemies can traverse normally.
    - The "1" cells represent the collision areas where sprites cannot go through and will have to change their direction when they reach it head on.

1.2. Initialization

- Binary map collision relies on the fact that the game world is a grid.
- Game object instances can "access" a new cell depending on that cell's value.
- Therefore, using a 2-dimensional array of "bools" to represent this 2D grid is perfectly suited for this problem, since each element in the 2D array will represent 1 cell.
- Divide the game world into a rectangular grid.
    - Or at least the "usable" part of the world because most worlds have areas around the edges that can only be seen and never reached by the sprites.
- Create two 2 dimensional arrays having a size equal to the rectangular grid.
- One array called "Map Data" will hold the initial positions of all the sprites and collision areas. (Collision areas, non-collision areas, main character, enemies, coins...)
- The other array called "Collision Data" will just hold the collision data.

    - Example:

Map Data

1	1	1	1	1
1	0	0	0	1
1	4	2	0	1
1	1	1	3	1
1	1	1	1	1

Collision Data

1	1	1	1	1
1	0	0	0	1
1	0	0	0	1
1	1	1	0	1
1	1	1	1	1

  - "Map Data" is generally imported from a file which has been previously exported from an editor. Then "Collision Data" is constructed using "Map Data".
  - The example represents an array that divides the world into a 5 by 5 grid.
  - Since each cell, whose value is 1 is a collision area, it is clear that the world is surrounded by collision areas in this example.
  - A small platform (highlighted in red) is placed in the bottom left corner of the world (or the top left depending on how the map was exported from the map editor. More on that later).
  - This array can be static for games like Pacman where the map does not change during game play.
  - If a platform game has destroyable bridges, then it is the programmer's responsibility to update the collision array appropriately.

1.3. Point collision

- At run time, if you want to check if a point is in a “solid” cell, get its cell position (which means getting the indices of the array element that represents that cell) and check its value.
- Example:
  - The cells' dimensions are (1; 1) each.
  - A point is located at position (3.3; 2.7) in the map coordinates system (more on that in the Map Normalized Coordinates System section)
  - Point_X = 3.3   Floor(3.3) = 3
  - Point_Y = 2.7   Floor(2.7) = 2
  - Fetch the value of the element [3][2] in our 2-dimensional array.
  - If it is 1, then this point is located in a solid area (The sprite has collided with a wall or a platform...)
  - If it is 0, then this point is not in a solid area, but this does not mean that the entire sprite isn't. Remember that other points or parts of the sprites may be inside some solid areas. (More on that in the hot spots section)

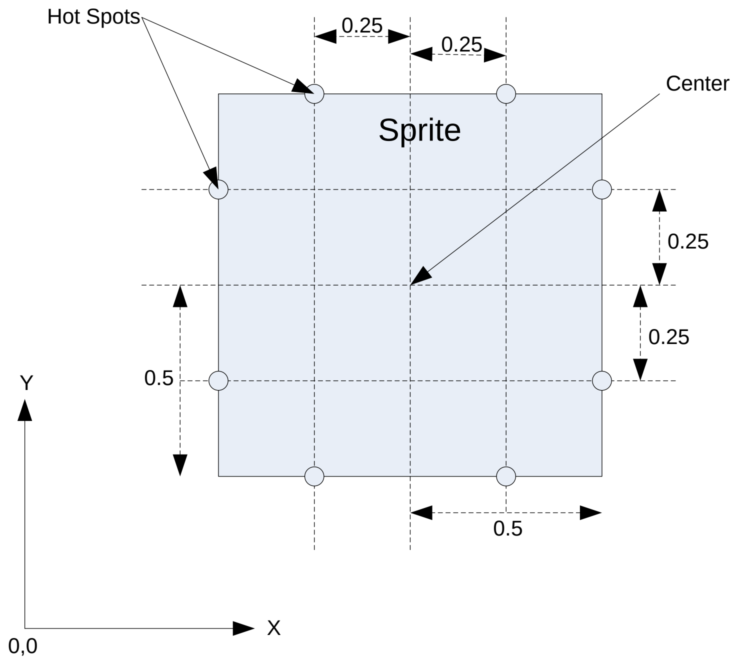
1.4. Sprite collision using hot spots

- So far, we covered how to check if a point is inside a solid area or not.
- But in games, object instances are not just points. They can be squares, rectangles.
  - Generally, they are made from a group of triangles.
- So how can we check for collision between an object instance and the binary collision map?
- Basically, the same concept applies: Test for collision between a point and the map.
- But *each object instance* has more than 1 point to check.
- These points are called hot spots.
- Each loop, we will have to check for collision between all these hotspots and the binary collision map.
- Finally, we update the position of the object instance according to the result of the collision check.

- This method assumes that the width and height of the object instance are both 1 (an object instance is the same size as 1 cell of the grid).

- Hot spots can be positioned manually.
  - Each game object instance has an array of hot spots.
  - While initializing the level, place all the needed hotspots on the object instance's edges.
  - Test for collision between these hot spots and the binary collision map at run time.

- Hot spots positions can be computed at run time.
  - When checking for collision between a certain object instance and the binary collision map, a loop will be used to compute the hot spots positions.
  - For each hot spot, we check if it resides in a collision area or not.
  - The trick is to have the smallest number of hotspots (in order to reduce the number of collision checks), but still be able to avoid having the object instance being partially inside a collision area.
  - There are several methods to get a good selection of hot spots.
  - We will cover one of them in this course.
    - Remember that the object instance's width and height are both equal to 1.
    - The algorithm will place 2 hot spots on each side of the object instance.
    - These hot spots are positioned on the sides, halfway between the middle of the side and its corners.



- For each object instance, we generate these hot spots, and then we check for collision between them and the binary collision map.

- We should determine if the object instance is colliding from its left, right, top, or bottom side.
    - The collision could be occurring on more than 1 side.
        - Example: If the object instance is moving upward and to the right, it might collide from its top and right sides simultaneously.
    - When checking for collision, we should determine all the collision sides.
    - Each game object instance will have a collision flag, where each bit represents one side.
        - The least significant bit represents the left side.
        - The second bit represents the right side.
        - The third bit represents the top side.
        - The fourth bit represents the bottom side.

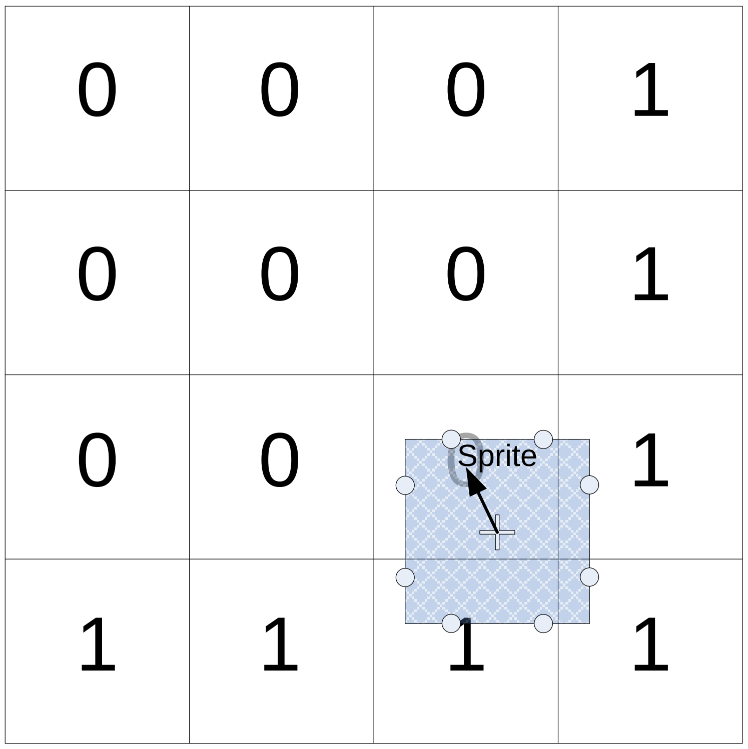
    - An enumeration of “defines” can be used for the sides' bits:

```
#define      COLLISION_LEFT             0x00000001    //0001
#define      COLLISION_RIGHT            0x00000002    //0010
#define      COLLISION_TOP              0x00000004    //0100
#define      COLLISION_BOTTOM           0x00000008    //1000
```

    - Each time a certain side is found in a collision state, we set its corresponding bit in the collision flag variable to 1.
    - This done by ORing the collision flag with the correspondent collision side value.
    - At the end of the collision check, the flag variable can be used to determine which sides of the object instance have collided with solid areas.
    - This is done by ANDing the collision flag with the correspondent collision side value.

1.5. Snapping

- After checking where each hot spot is, we should snap back the object instance in case it was in a collision area.
    - It is considered colliding with a collision area if at least 1 hot spot is inside a collision area, which means the collision flag has one or more bits set to 1.
- If at least one of the left or right hot spots is inside a collision area, we should snap the X position of the corresponding sprite back to the middle of its center's cell.
- Similarly, if at least one of the top or bottom hot spots is inside a collision area, we should snap the Y position of the corresponding sprite back to the middle of its center's current cell.



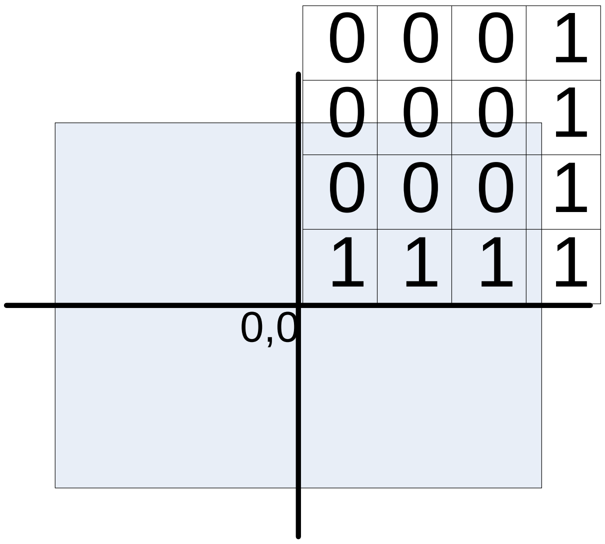
PosX=(int)(PosX)+0.5

PosY=(int)(PosY)+0.5

- Changing the position of the sprite that way will make sure that it is not colliding with a collision area anymore.
- Note that this position snapping is done before rendering the entire scene; therefore, the object will never be rendered when it's partially inside the collision area.

1.6. Normalized coordinates system

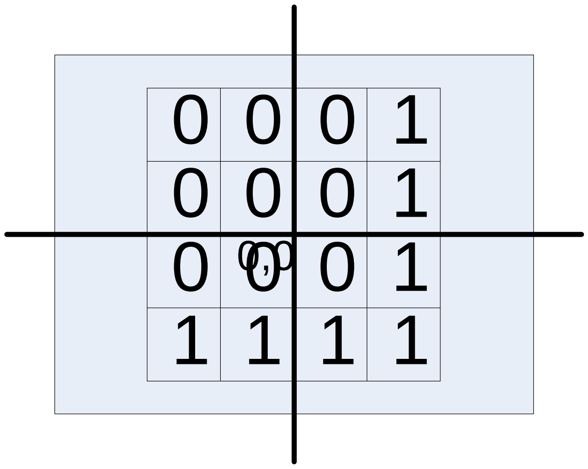
- Visually, a cell in a binary map can have its width different than its height.
    - The art assets might require that feature.
    - We might want to scale the entire map to the size on the window, regardless of the grid's width and height.
- Logically, the cell's width and height should directly affect the collision check, since each point we want to check, will have to find its corresponding cell position.
- To avoid this issue, we should have the binary map normalized.
- Basically, the width and height of each cell in the normalized coordinates system are 1.0, independently from the final visual result (even if visually, the width of a cell is different than its height).
- All the objects instances are within the normalized coordinates system of the binary map.
    - Their positions are in that normalized coordinates system.
    - Example:
    - (0, 0): Represents the bottom left of the bottom left cell.
    - (1, 1): Represents the top right corner of the bottom left cell.
    - (2, 3.5): Represents the right side of the second column of cells and 3 and half cells upward.
    - Since the position of an object instance is derived from its velocity, then consequently, its velocity vector is also relative to the normalized coordinates system of the binary collision map.
    - Example:
        - A velocity of (1;0) will move the object 1 cell to the right each loop (or each second if it is time based).
    - Also, since the velocity of *some* objects instances is derived from their acceleration, then the acceleration is relative to the normalized coordinates system of the binary collision map too.
- All the physics, movement and collision checks are done in this normalized coordinates system.
- After computing the physics movement, checking for collision, and snapping the object instances to the correct locations, we will have to transform them out of the normalized coordinates system of the binary map before actually rendering them.
- This transformation is usually made from 2 transformations (but it does not have to): A translation and a scale.
    - The translation is needed to translate all the objects instances (and the tiles in case they exist) to the correct position in the world.
        - Remember that in the normalized coordinates system, the bottom left of the binary collision map is considered (0, 0).
        - And we know that (0, 0) usually gets rendered in the middle of our window.
        - If we do not apply a translation, we will have the bottom left of the grid positioned in the middle of the screen.



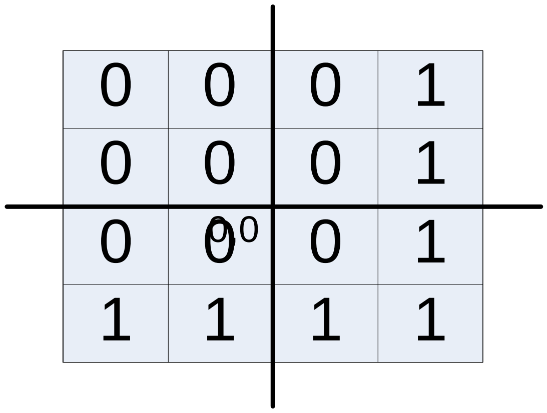
- Our goal is to have the *middle* of the grid positioned in the middle of the screen.
- Basically, we will have to translate the grid and all the objects instances by the negative of half the grid's width and height.

$$\begin{bmatrix} 1 & 0 & \frac{-\,Grid\ Width}{2} \\ 0 & 1 & \frac{-\,Grid\ Height}{2} \\ 0 & 0 & 1 \end{bmatrix}$$

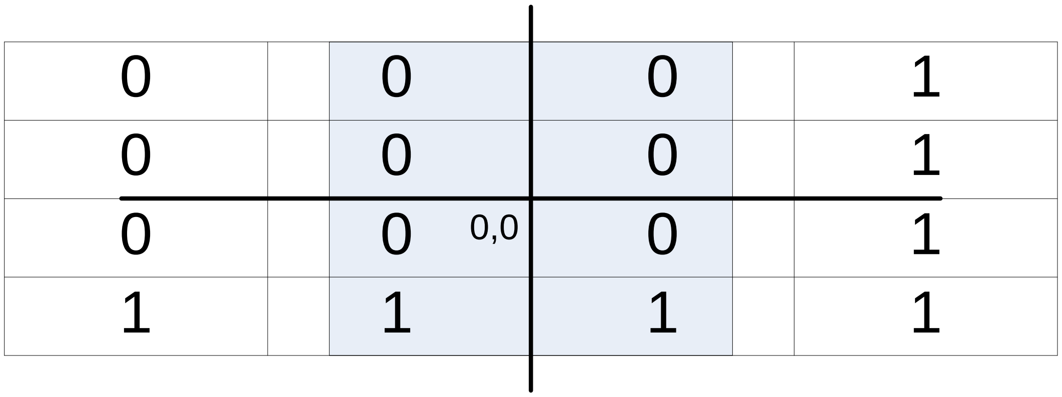
- After applying a translation matrix, we will have to compute a scaling matrix.
  - Usually, we will have to scale everything up from the normalized coordinates system of the binary map.



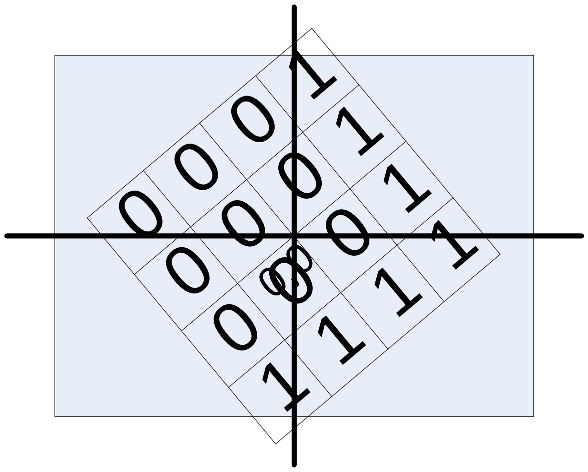
- One option would be to determine a certain scale value to make the entire grid have the same size of our window.



- Another option would be to use a scaling values that make the size of the grid greater than the view port, which makes a portion of it visible while the other is not (Good for scrolling games).



- The map can also be rotated. Keep in mind that collision checks are not affected at all by these transformations, since they were done in the map's normalized coordinates system.



1.7. Flipping the Y value

- In most games, if a point's Y coordinate increases, it moves upwards.
- On the other hand, editors are usually implemented using the window's coordinates system, where the origin is the top left corner. This means that if a point's Y coordinate increases, it moves downwards.
- This is a problem for games that import map data from editors.

  Example:
- Step 1: Creating the above map using the map editor:

	0	1	2	3	4
0	1	1	1	1	1
1	1	0	0	0	1
2	1	4	2	0	1
3	1	1	1	3	1
4	1	1	1	1	1

- Notice that the top left element (cell) in the map is actually the one indexed at [0][0] in memory

- Step #2: Importing the map
  - Importing the map created in step #1 without flipping the data on the Y axis will yield the following in-game map:

4	1	1	1	1	1
3	1	1	1	3	1
2	1	4	2	0	1
1	1	0	0	0	1
0	1	1	1	1	1
	0	1	2	3	4

  - Obviously, this is not the desired result. (Everything is flipped!)
  - **This is due to the fact that the coordinates system changed while the data didn't.**

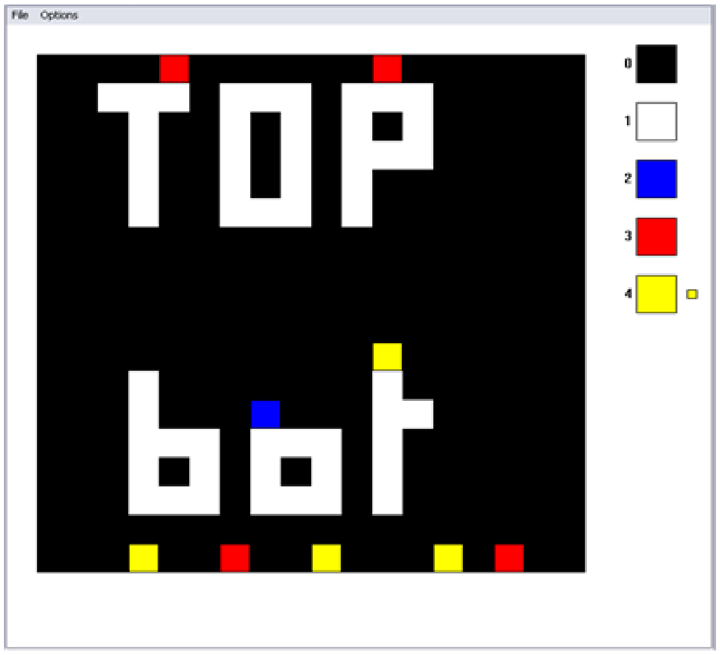
- To fix this, we need to flip the data on the Y axis, which can be done in multiple techniques:
  - Method #1: Flip the data internally in the editor.
  - Method #2: Flip the data upon exporting it from the editor.
  - Method #3: Flip the data while importing it.

4	1	1	1	1	1
3	1	0	0	0	1
2	1	4	2	0	1
1	1	1	1	3	1
0	1	1	1	1	1
	0	1	2	3	4

  - **In our project, we will use method #3**

1.8. Recapitulation

- Build a map using the editor



- Export the data to a file.
  Width 20
  Height 20

```
1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
1 0 0 0 0 3 0 0 0 0 0 0 3 0 0 0 0 0 0 1
1 0 0 1 1 1 0 1 1 1 0 1 1 1 0 0 0 0 0 1
1 0 0 0 1 0 0 1 0 1 0 1 0 1 0 0 0 0 0 1
1 0 0 0 1 0 0 1 0 1 0 1 1 1 0 0 0 0 0 1
1 0 0 0 1 0 0 1 0 1 0 1 0 0 0 0 0 0 0 1
1 0 0 0 1 0 0 1 1 1 0 1 0 0 0 0 0 0 0 1
1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1
1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1
1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1
1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1
1 0 0 0 0 0 0 0 0 0 0 0 4 0 0 0 0 0 0 1
1 0 0 0 1 0 0 0 0 0 0 0 1 0 0 0 0 0 0 1
1 0 0 0 1 0 0 0 2 0 0 0 1 1 0 0 0 0 0 1
1 0 0 0 1 1 1 0 1 1 1 0 1 0 0 0 0 0 0 1
1 0 0 0 1 0 1 0 1 0 1 0 1 0 0 0 0 0 0 1
1 0 0 0 1 1 1 0 1 1 1 0 1 0 0 0 0 0 0 1
1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1
1 0 0 0 4 0 0 3 0 0 4 0 0 0 4 0 3 0 0 1
```

- Import the data into the game while flipping the data on the Y axis.

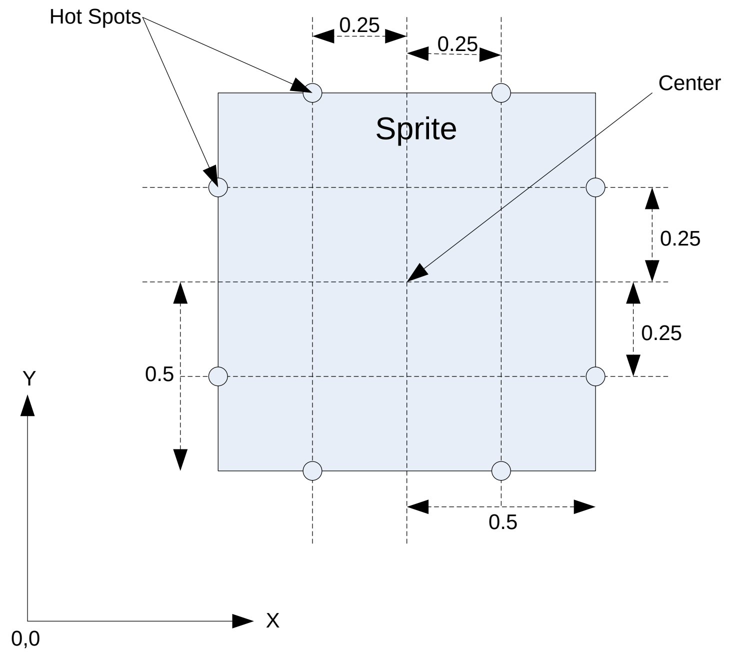
```
for y = (height – 1) to 0
         for x = 0 to (width – 1)
                      Value = Get next value from file
                      //Map data
                      Data[x][y] = Value

                      //Collision data
                      if(Value == 1)
                                  BinaryCollision[x][y] = 1
                      else
                                  BinaryCollision[x][y] = 0
```

- Reading the collision data

  Reading point (X, Y)

```
//Make sure X and Y are not out of bounds
//Get the value from the collision array
//Return the value
```

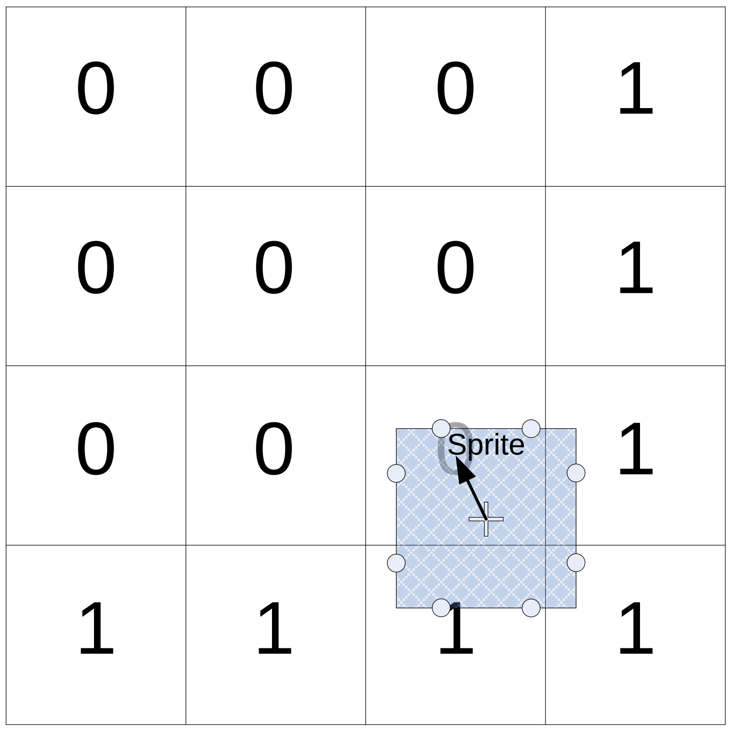
- Sprite collision using hot spots offset.
  - Determining hot spots positions



- Check which hot spots are in a collision area.
- Update the collision flag by setting the corresponding bit to 1.

```
if one of the top hot spots is in a collision area
    Set the correspondent bit to 1 // Flag |= TOP

if one of the bottom hot spots is in a collision area
    Set the correspondent bit to 1 // Flag |= BOTTOM

if one of the left hot spots is in a collision area
    Set the correspondent bit to 1 // Flag |= LEFT

if one of the right hot spots is in a collision area
    Set the correspondent bit to 1 // Flag |= RIGHT
```

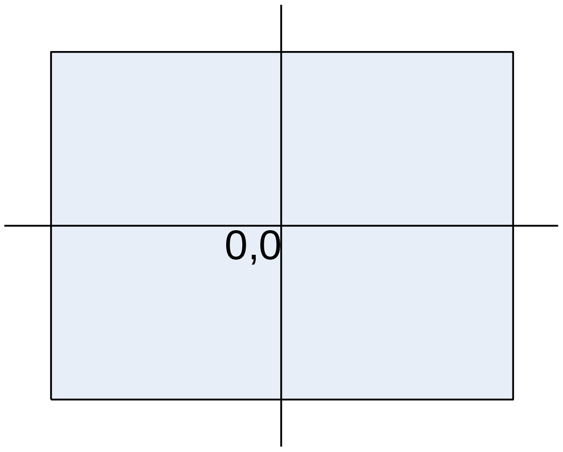
- Snapping the sprite
  - Reposition the sprite to the middle of the cell where its center lies.

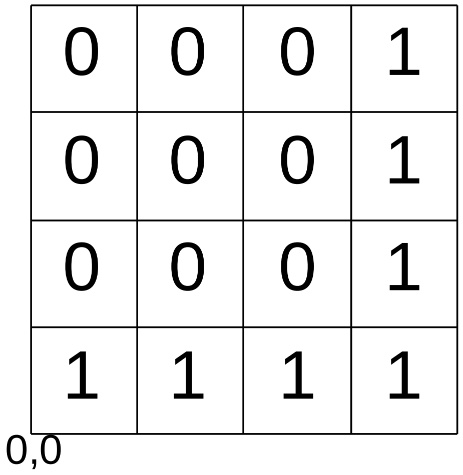


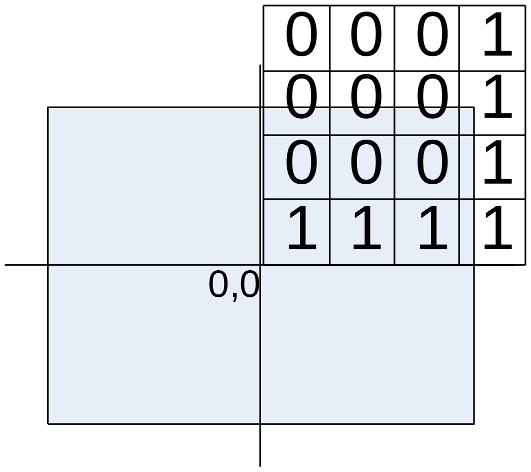
PosX=(int)(PosX)+0.5

PosY=(int)(PosY)+0.5

- A transformation is needed to transform sprites and tiles (if they exist) out of the normalized coordinates system of the binary map.
    - (0,0) represents the middle of the screen, but for the binary map, (0,0) represents the bottom left.

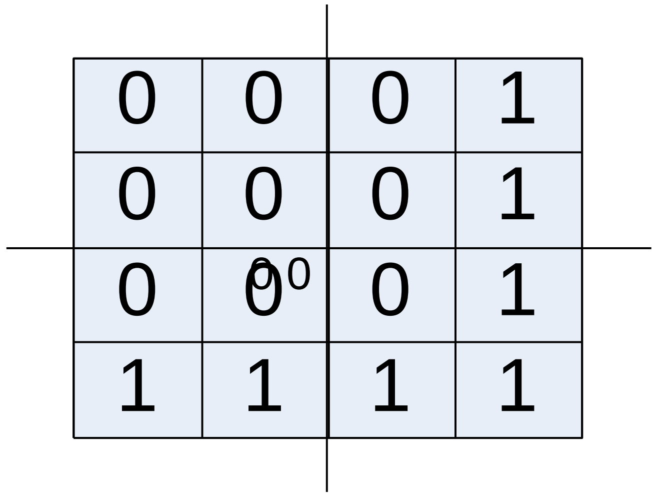




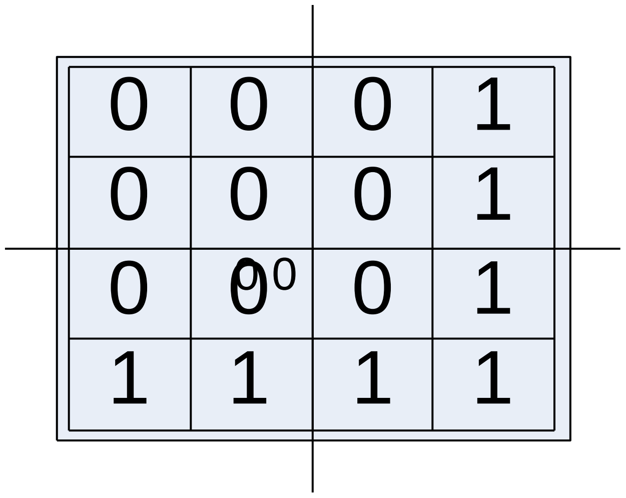


- We must translate by half the binary map's width and height.
- After translating, a scaling transformation is needed to set the size of the map and sprites to any desired value.

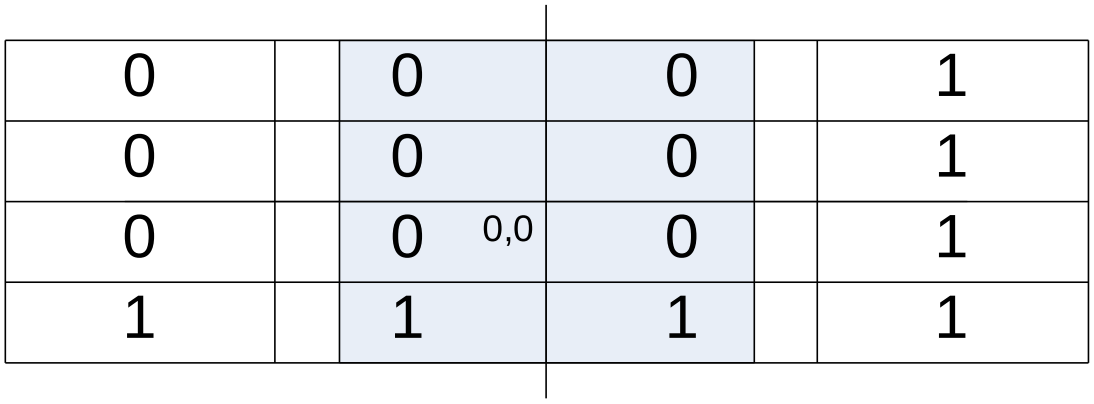
- Example 1: Scaling to a greater size



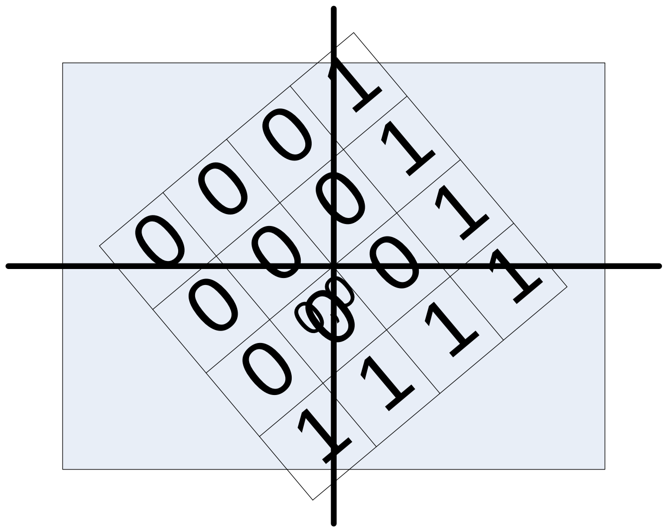
- Example2: Scaling to the size of the window



- Example3: Scaling to a size greater than the window (Scrolling game)



- Example 4: Rotate the map


0 0 0 1
0 0 0 1
0 0 0 1
0 0 1 1
1 1 1 1