Assignment 1

Done by German Devchich, 1-st year of Master degree ,Skoltech

2024 Moscow

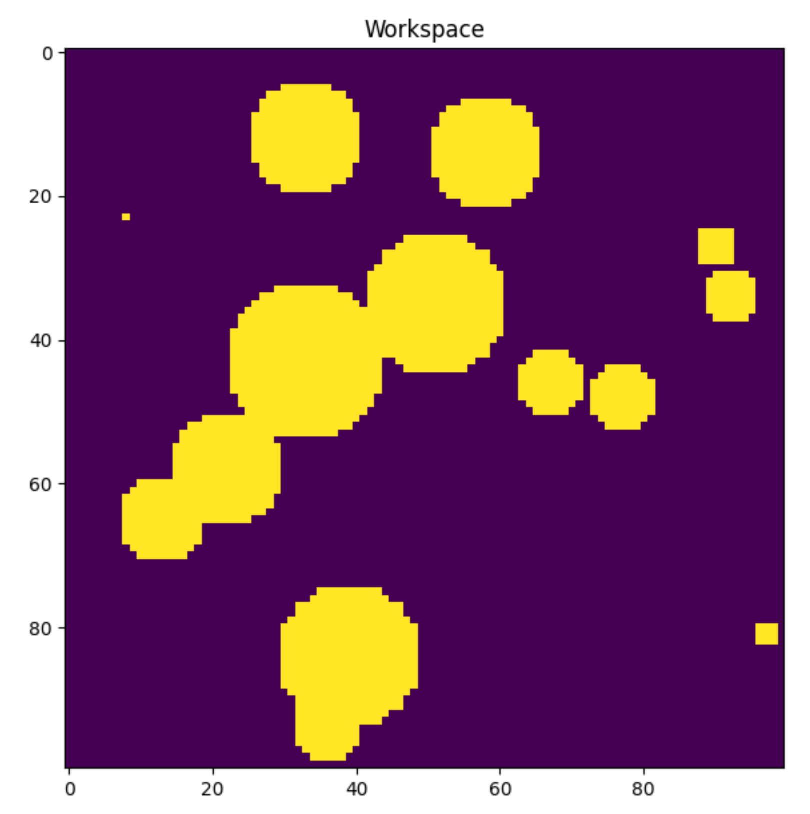
Task 1: Configuration Space (40 points)

In [1]:
```python
from utils import *
import numpy as np
from scipy.signal import convolve2d
from queue import PriorityQueue
from dataclasses import dataclass
```

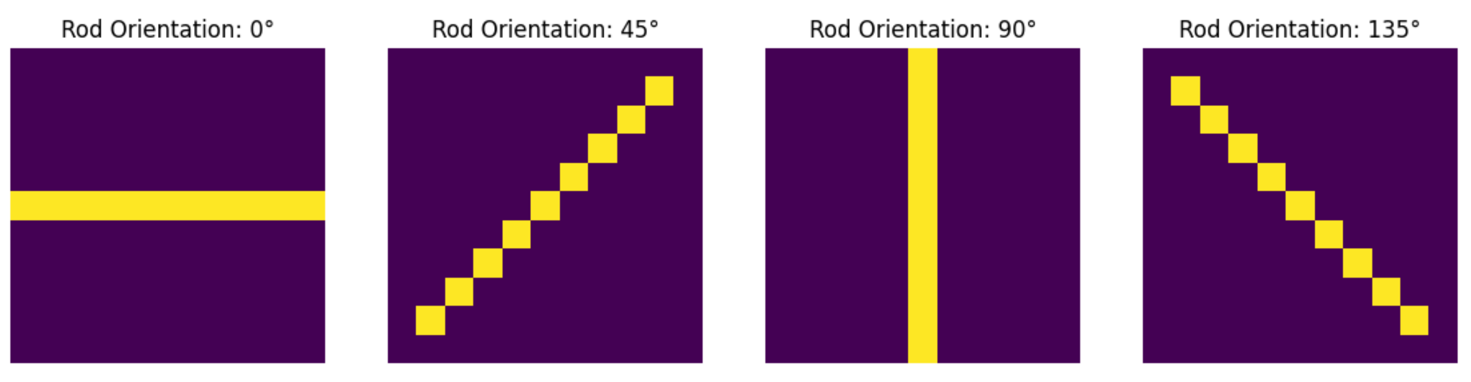
A (10 pts) Visualize from the given data the workspace and the different rod configurations for each discretized orientation. Comment on the given discretized values for orientation

In [2]:
```python
img = np.load('data_ps1.npz')
env = img['environment']
rod = img['rod']
```

In [3]:
```python
plt.figure(figsize=(14, 7))
plt.title("Workspace")
plt.imshow(env)
plt.show()
```



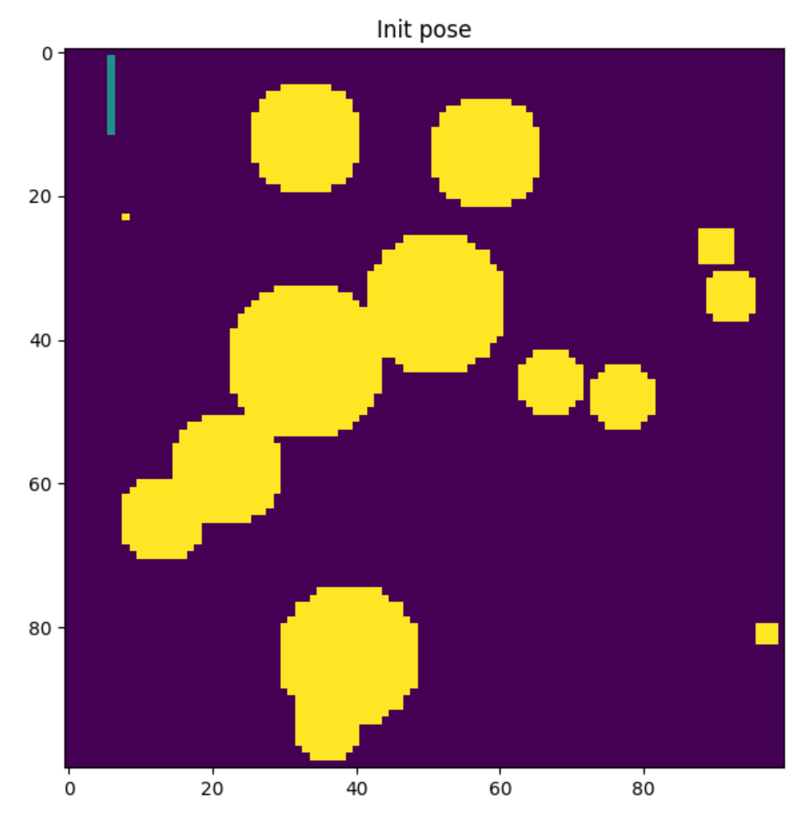
In [4]:
```python
orientations = ["0°", "45°", "90°", "135°"]
fig, axs = plt.subplots(1, 4, figsize=(14, 7))
for i in range(4):
    axs[i].imshow(rod[:, :, i])
    axs[i].set_title(f"Rod Orientation: {orientations[i]}")
    axs[i].axis('off')
plt.show()
```



The orientation of the robot is discretized with four angles: 0°, 45°, 90°, and 135°. This approach limits the possible angles the robot can take, but makes the calculations more efficient. For example, if the robot could rotate every degree, the size and complexity of the C-space would increase dramatically, making the computation less convenient.

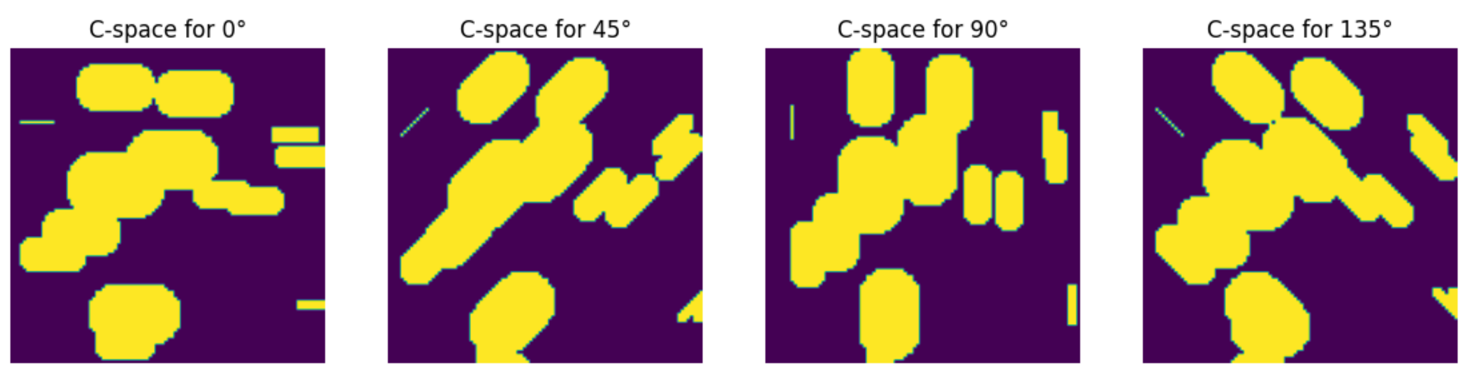
B (10 pts) Visualize the environment together with the object. For this, you may want to use the function **plot_joint_environment** from **utils.py** and select any valid configuration value for the rod.

In [5]:
```python
plt.figure(figsize=(14, 7))
plt.title("Init pose")
plt.imshow(plot_enviroment(img=env,obj=rod,state=(6,6,2)))
plt.show()
```



C (10 pts) Create the C-space for the 2D environment map. For this, plot all the images corresponding to each of the orientations by using collision checking.

In [6]:
```python
c_space = np.array([normalize_image(convolve2d(in1=env,in2=rod[:, :, i],boundary='symm', mode='same')) for i in range(4)])
orientations = ["0°", "45°", "90°", "135°"]
fig, axs = plt.subplots(1, 4, figsize=(14, 7))
for i in range(c_space.shape[0]):
    axs[i].imshow(c_space[i])
    axs[i].set_title(f"C-space for {orientations[i]}")
    axs[i].axis('off')
plt.show()
```



D (10 pts) Comment on the obtained C-space with the previous method. What is the size of the C-space?

The created C-space for each robot orientation (four layers for 0°, 45°, 90° and 135°) has a dimension of 4, 100,100. It displays where the robot can be located while avoiding collisions.

To build this space, a convolution with a kernel in the form of the position of our robot is used to determine the presence of obstacles for each robot orientation at each point on the map.

This space is subsequently used in the pathfinding algorithm.

Task 2: A star Algorithm (60 points)

A (40 pts) You need to implement the A star algorithm and plan in the generated discrete C-space from the previous task. The starting configuration of the agent is $(6, 6, 2)$ and the goal configuration is $(72, 64, 0)$. On this first iteration, use an heuristic function $h(q, q_G) = 0$, which is equivalent to the Dijkstra algorithm.

Save the result of calculated plan in **rod_solve.mp4** using **plotting_result(environment,rod,plan)** from **utils.py**, where plan is list of rod states from start to goal.

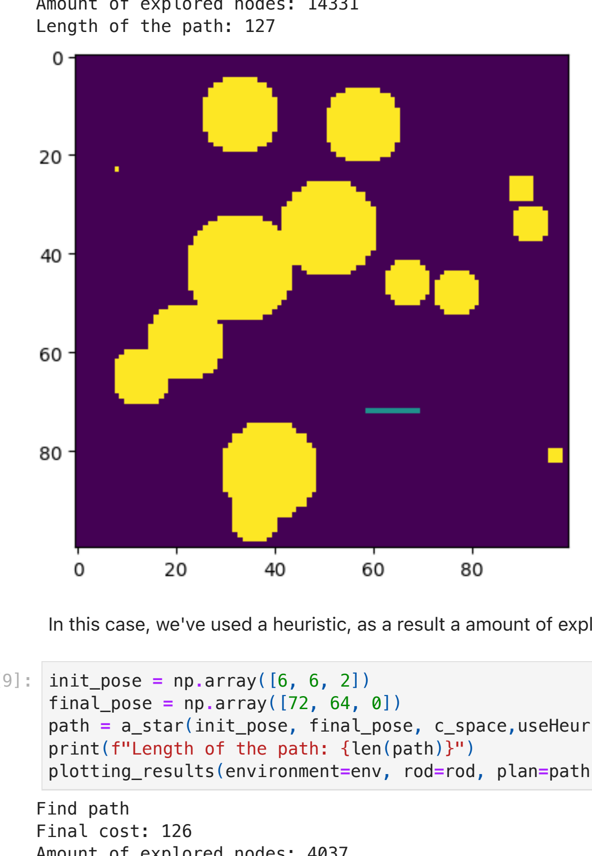
hint: Track the number of visited states to avoid/debug potential issues with internal loops

In [7]:
```python
def a_star(init_pose: np.ndarray, final_pose: np.ndarray, env_map: np.ndarray, useHeuristic=False, useAngle=False):

    num_layers = env_map.shape[0]

    xy_cost = 1
    rot_cost = 1


    def heuristic(x1, x2):
        if useAngle:
            angle_difference = abs(x1[2]-x2[2]) % num_layers
            return np.linalg.norm(x1[:2] - x2[:2]) + min(angle_difference, num_layers - angle_difference)
        if useHeuristic:
            return abs(int(x1[0]) - int(x2[0])) + abs(int(x1[1]) - int(x2[1]))
        else:
            return 0

    def state_evolution(x_current, cost_current):
        transitions = np.array([
            [1, 0, 0, xy_cost],
            [-1, 0, 0, xy_cost],
            [0, 1, 0, xy_cost],
            [0, -1, 0, xy_cost],
            [0, 0, 1, rot_cost],
            [0, 0, -1, rot_cost]
        ])

        future_states = []
        for transition in transitions:
            prob_x = x_current + transition[:3]
            cost_future = transition[3]
            prob_x[2] %= num_layers  # normalizing angle

            x = int(prob_x[0])
            y = int(prob_x[1])
            theta = int(prob_x[2])

            # calculating the bounds of obj
            dim_y = int((rod.shape[0] - 1) / 2)
            dim_x = int((rod.shape[1] - 1) / 2)

            y_min = y - dim_y
            y_max = y + dim_y + 1
            x_min = x - dim_x
            x_max = x + dim_x + 1

            # Check that obj in bounds of the map
            if y_min < 0 or x_min < 0 or y_max > env_map.shape[1] or x_max > env_map.shape[2]:
                continue

            # Check collision in c_space
            if env_map[theta, x, y] == 0:
                future_states.append((prob_x, cost_future))
        return future_states

    q = PriorityQueue()
    init_pose_tuple = tuple(map(int, init_pose))
    q.put((heuristic(init_pose, final_pose), init_pose_tuple))

    came_from = {}
    c_cost = {init_pose_tuple: 0}

    already_visited = set()
    nodes_expanded = 0

    while not q.empty():
        _, x_current_tuple = q.get()
        x_current = np.array(x_current_tuple)

        if x_current_tuple in already_visited:
            continue
        already_visited.add(x_current_tuple)
        nodes_expanded += 1
        if np.array_equal(x_current, final_pose):
            plan = []
            while x_current_tuple in came_from:
                plan.append(x_current)
                x_current_tuple = came_from[x_current_tuple]
                x_current = np.array(x_current_tuple)
            plan.append(init_pose)
            print("Find path")
            final_cost = c_cost[tuple(map(int, final_pose))]
            print(f"Final cost: {final_cost}")
            print(f"Amount of explored nodes: {nodes_expanded}")
            standardized_plan = [tuple(map(int, state)) for state in plan[::-1]]
            return standardized_plan

        for prob_x, cost_future in state_evolution(x_current, c_cost[x_current_tuple]):
            prob_x_tuple = tuple(map(int, prob_x))

            tentative_c_cost = c_cost[x_current_tuple] + cost_future

            if prob_x_tuple not in c_cost or (tentative_c_cost < c_cost[prob_x_tuple]):
                came_from[prob_x_tuple] = x_current_tuple
                c_cost[prob_x_tuple] = tentative_c_cost
                total_cost = tentative_c_cost + heuristic(prob_x, final_pose)

                q.put((total_cost, prob_x_tuple))

    print("No way")
    return None
```

In this case we don't use a heuristic, as a result our algorithm works as a Dijkstra's algorithm, due to this fact we explore much more nodes comparing with A^*

In [8]:
```python
init_pose = np.array([6, 6, 2])
final_pose = np.array([72, 64, 0])
path = a_star(init_pose, final_pose, c_space,useHeuristic=False)
print(f"Length of the path: {len(path)}")
plotting_results(environment=env, rod=rod, plan=path, save_path='rod_solve_Dj.mp4')
```

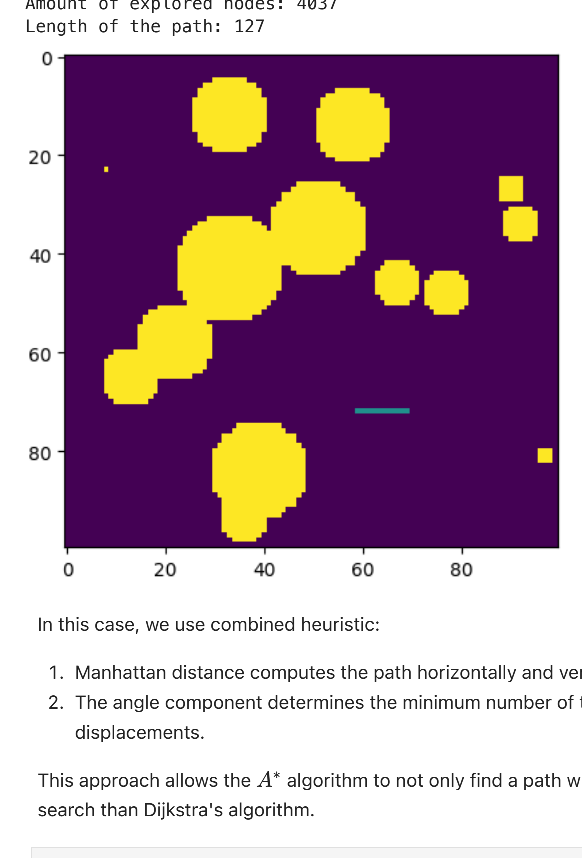
```
Find path
Final cost: 126
Amount of explored nodes: 14331
Length of the path: 127
```



In this case, we've used a heuristic, as a result a amount of explored decreases.

In [9]:
```python
init_pose = np.array([6, 6, 2])
final_pose = np.array([72, 64, 0])
path = a_star(init_pose, final_pose, c_space,useHeuristic=True)
print(f"Length of the path: {len(path)}")
plotting_results(environment=env, rod=rod, plan=path, save_path='rod_solve_A.mp4')
```

```
Find path
Final cost: 126
Amount of explored nodes: 4037
Length of the path: 127
```



In this case, we use combined heuristic:

1. Manhattan distance computes the path horizontally and vertically, which is efficient in a discrete grid.
2. The angle component determines the minimum number of turns required to reach the desired orientation layer. This component is especially useful if turns are more “expensive” than displacements.

This approach allows the A^* algorithm to not only find a path with a minimum number of displacements, but also minimize the number of turns, resulting in a faster and more efficient path search than Dijkstra's algorithm.

In [10]:
```python
init_pose = np.array([6, 6, 2])
final_pose = np.array([72, 64, 0])
path = a_star(init_pose, final_pose, c_space,useAngle=True)
print(f"Length of the path: {len(path)}")
plotting_results(environment=env, rod=rod, plan=path, save_path='rod_sole_angle.mp4')
```

```
Find path
Final cost: 126
Amount of explored nodes: 4520
Length of the path: 127
```