In [1]:
```python
import pandas as pd
import numpy as np
import math
from scipy import stats
from scipy.stats import norm
from random import choices

import requests

import plotly.graph_objects as go

from matplotlib import pyplot as plt
from matplotlib.patches import Circle, Rectangle, Arc
%matplotlib inline

from sklearn import linear_model
from sklearn.linear_model import LinearRegression

import statsmodels.api as sm
```

In [2]:
```python
path = "NBA_2004_2023_Shots.csv"
```

In [3]:
```python
data_shots = pd.read_csv(path)
#data_shots2 = pd.read_csv(path) #  The next steps will require the origi
```

In [4]:
```python
# This ensures that when printing the Dataframe, all of the columns will
pd.set_option('display.max_columns', None)
```

In [5]:
```python
# Condition for dropping '2PT Field Goal' with distance > 28
condition_2pt = (data_shots['SHOT_TYPE'] == '2PT Field Goal') & (data_sho
# Condition for dropping '3PT Field Goal' with distance < 22
condition_3pt = (data_shots['SHOT_TYPE'] == '3PT Field Goal') & (data_sho

# Combine conditions
condition_to_drop = condition_2pt | condition_3pt

# Drop rows matching the condition
data_shots = data_shots[~condition_to_drop]
```

In [6]:
```python
# Create a mask for shots made
shots_made_mask = data_shots['SHOT_MADE']

# Initialize POINTS column with 0
data_shots['POINTS'] = 0

# Assign points based on the shot type for shots made
data_shots.loc[shots_made_mask & (data_shots['SHOT_TYPE'] == '2PT Field G
data_shots.loc[shots_made_mask & (data_shots['SHOT_TYPE'] == '3PT Field G
```

In [7]:
```python
# Define categories for shots
categories = {
    'Layups': [
        'Layup Shot', 'Finger Roll Layup Shot', 'Floating Layup Shot', 'R
        'Driving Layup Shot', 'Cutting Layup Shot', 'Putback Layup Shot',
        'Running Finger Roll Layup Shot', 'Running Reverse Layup Shot', '
        'Driving Reverse Layup Shot', 'Cutting Finger Roll Layup Shot', '
```

```
        'Driving Finger Roll Shot', 'Turnaround Finger Roll Shot', 'Runni
        'Driving Reverse Layup Shot', 'Tip Layup Shot'
    ],
    'Dunks': [
        'Dunk Shot', 'Alley Oop Dunk Shot', 'Cutting Dunk Shot', 'Driving
        'Putback Dunk Shot', 'Running Dunk Shot', 'Tip Dunk Shot', 'Slam 
        'Driving Reverse Dunk Shot', 'Running Reverse Dunk Shot', 'Putbac
        'Reverse Slam Dunk Shot', 'Running Slam Dunk Shot', 'Putback Reve
        'Tip Dunk Shot'
    ],
    'Jump Shots': [
        'Jump Shot', 'Pullup Jump Shot', 'Fadeaway Jump Shot', 'Running J
        'Step Back Jump Shot', 'Turnaround Jump Shot', 'Floating Jump Sho
        'Running Pull-Up Jump Shot', 'Driving Floating Jump Shot', 'Drivi
    ],
    'Hook Shots': [
        'Hook Shot', 'Driving Hook Shot', 'Turnaround Hook Shot', 'Turnar
        'Hook Bank Shot', 'Running Hook Shot', 'Jump Hook Shot', 'Running
        'Jump Bank Hook Shot'
    ],
    'Bank Shots': [
        'Jump Bank Shot', 'Fadeaway Bank Shot', 'Turnaround Bank Shot',
        'Driving Bank Hook Shot', 'Step Back Bank Jump Shot', 'Turnaround
        'Driving Bank shot', 'Pullup Bank shot', 'Running Bank shot'
    ],
    'Tip-ins': [
        'Tip Shot', 'Running Tip Shot'
    ],
    'No_Shot' : [
        'No Shot'
    ]

}

# Function to categorize the shots
def categorize(categories, action_type):
    action_type = action_type.lower().strip()
    for category, types in categories.items():
        if any(action_type == t.lower().strip() for t in types):
            return category
    return 'other'

# Apply categorization
data_shots['SHOT_CATEGORY'] = data_shots['ACTION_TYPE'].apply(lambda x: c
data_shots2 = data_shots.copy()
```

In [8]:

```
# Function for convert the feet in meters
def feet_to_m(x):
  out = round(x*0.3048, 2)
  return out

data_shots['LOC_X'] = data_shots['LOC_X'].apply(lambda x: feet_to_m(x))
data_shots['LOC_Y'] = data_shots['LOC_Y'].apply(lambda x: feet_to_m(x))
```

In [9]:

```
# Calculate angle from center for each shot
data_shots['ANGLE_FROM_CENTER'] = np.arctan2(data_shots['LOC_Y'], data_sh

# Adjust angles for shots in the left half (x < 0)
left_half_mask = data_shots['LOC_X'] < 0
```

```
data_shots.loc[left_half_mask, 'ANGLE_FROM_CENTER'] += np.pi

# Convert angles to degrees if needed
data_shots['ANGLE_FROM_CENTER_DEGREES'] = np.degrees(data_shots['ANGLE_FR
```

In [10]:
```
data_shots
```

Out[10]:

	SEASON_1	SEASON_2	TEAM_ID	TEAM_NAME	PLAYER_ID	PLAYER_I
0	2023	2022-23	1610612764	Washington Wizards	203078	Bradle
1	2023	2022-23	1610612764	Washington Wizards	204001	Kr Por
2	2023	2022-23	1610612764	Washington Wizards	1628420	Monte I
3	2023	2022-23	1610612764	Washington Wizards	204001	Kr Por
4	2023	2022-23	1610612764	Washington Wizards	1630166	Deni
...	...	...	...	...	...	
4012556	2004	2003-04	1610612755	Philadelphia 76ers	2422	John Sa
4012557	2004	2003-04	1610612759	San Antonio Spurs	1938	Manu G
4012558	2004	2003-04	1610612747	Los Angeles Lakers	406	Sha C
4012559	2004	2003-04	1610612756	Phoenix Suns	2063	Jake Vo
4012560	2004	2003-04	1610612748	Miami Heat	2548	Dwyane

4012098 rows × 30 columns

In [11]:
```
# Select the season to analyse
season_1 = data_shots[data_shots["SEASON_1"] == (2006)].copy()
season_2 = data_shots[data_shots["SEASON_1"] == (2006)].copy()
```

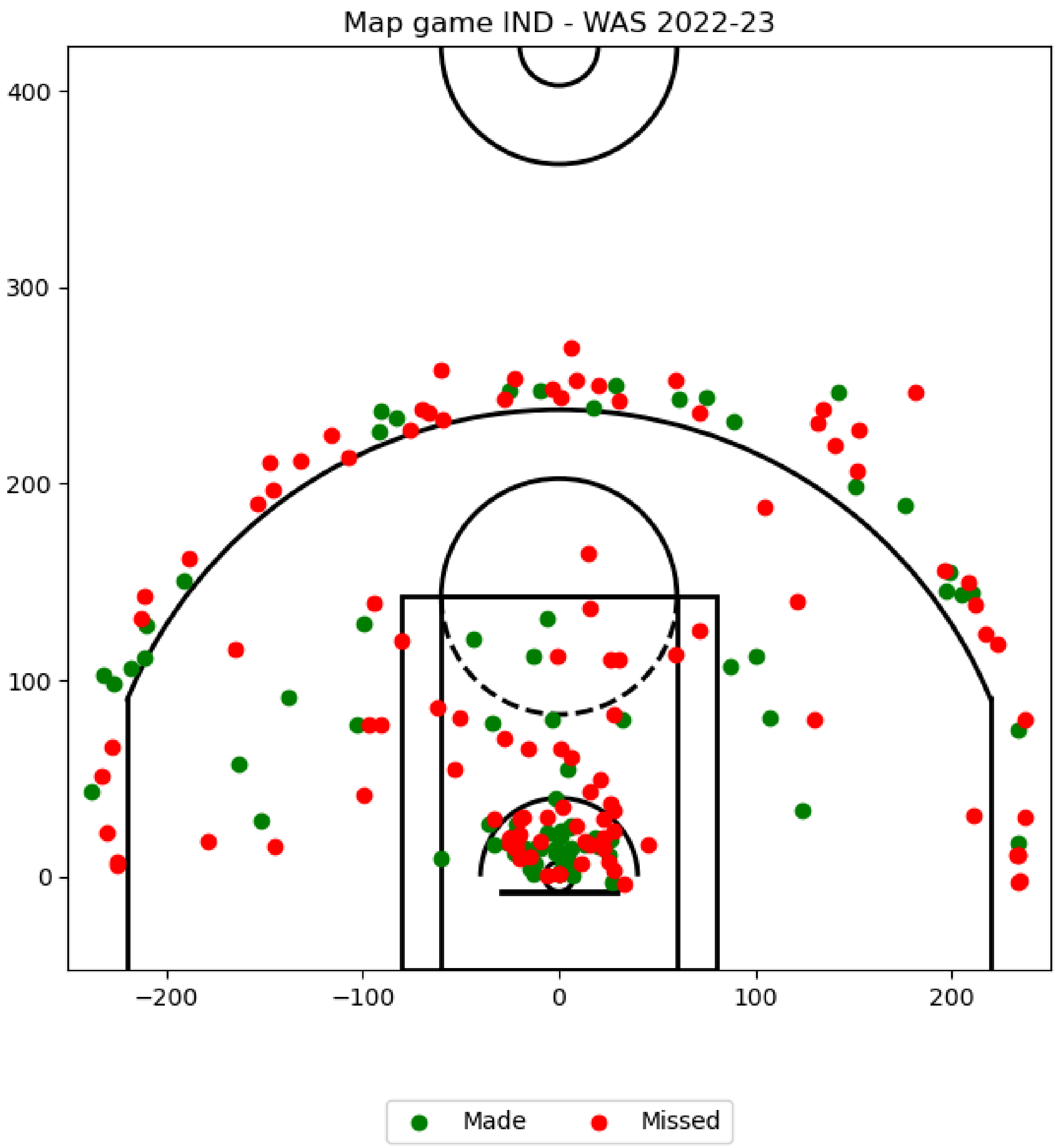
PLOTTING

In [12]:
```
def draw_court(ax=None, color='black', lw=2, outer_lines=False, interval=
    if ax is None:
        ax = plt.gca()

    # Create the basketball hoop
    hoop = Circle((0, 0), radius=7.5, linewidth=lw, color=color, fill=Fal

    # Create backboard
    backboard = Rectangle((-30, -7.5), 60, -1, linewidth=lw, color=color)

    # The paint
    # Create the outer box 0f the paint, width=16ft, height=19ft
```

```
    outer_box = Rectangle((-80, -47.5), 160, 190, linewidth=lw, color=col
                          fill=False)
    # Create the inner box of the paint, widt=12ft, height=19ft
    inner_box = Rectangle((-60, -47.5), 120, 190, linewidth=lw, color=col
                          fill=False)

    # Create free throw top arc
    top_free_throw = Arc((0, 142.5), 120, 120, theta1=0, theta2=180,
                         linewidth=lw, color=color, fill=False)
    # Create free throw bottom arc
    bottom_free_throw = Arc((0, 142.5), 120, 120, theta1=180, theta2=0,
                            linewidth=lw, color=color, linestyle='dashed'
    # Restricted Zone, it is an arc with 4ft radius from center of the ho
    restricted = Arc((0, 0), 80, 80, theta1=0, theta2=180, linewidth=lw,
                     color=color)

    # Three point line
    # Create the side 3pt lines, they are 14ft long before they begin to
    corner_three_a = Rectangle((-220, -47.5), 0, 140, linewidth=lw,
                               color=color)
    corner_three_b = Rectangle((220, -47.5), 0, 140, linewidth=lw, color=
    # 3pt arc - center of arc will be the hoop, arc is 23'9" away from ho
    three_arc = Arc((0, 0), 475, 475, theta1=22, theta2=158, linewidth=lw
                    color=color)

    # Center Court
    center_outer_arc = Arc((0, 422.5), 120, 120, theta1=180, theta2=0,
                           linewidth=lw, color=color)
    center_inner_arc = Arc((0, 422.5), 40, 40, theta1=180, theta2=0,
                           linewidth=lw, color=color)

    court_elements = [hoop, backboard, outer_box, inner_box, top_free_thr
                      bottom_free_throw, restricted, corner_three_a,
                      corner_three_b, three_arc, center_outer_arc,
                      center_inner_arc]

    if outer_lines:
        # Draw the half court line, baseline and side out bound lines
        outer_lines = Rectangle((-250, -47.5), 500, 470, linewidth=lw,
                                color=color, fill=False)
        court_elements.append(outer_lines)

    for element in court_elements:
        ax.add_patch(element)


    ax.set_aspect('equal', adjustable='box')
    ax.set_xlim(-250, 250)
    ax.set_ylim(-47.5, 422.5)

    return ax
```

In [13]:
```
shots = pd.DataFrame()

game_id = 22200004

# Retrive the shots of the right match
shots = data_shots[(data_shots['GAME_ID']) == (game_id)]

# Separate the shots made from the shots failed
```

```
made_shots = shots[shots['SHOT_MADE'] == True]
missed_shots = shots[shots['SHOT_MADE'] == False]

# Take the right things for the title of the plot
team_home = (shots['HOME_TEAM']).iloc[0]
team_away = (shots['AWAY_TEAM']).iloc[0]
team_season = (shots['SEASON_2']).iloc[0]

# Setting the parameters for the plot
fig, ax = plt.subplots(figsize=(10, 7))

# Draw the lines field
draw_court(ax)

# Draw the made (green) and the failed (red)
ax.scatter((made_shots['LOC_X']/0.3048)*10, (made_shots['LOC_Y']/0.3048)*
ax.scatter((missed_shots['LOC_X']/0.3048)*10, (missed_shots['LOC_Y']/0.30

# Title of the plot
ax.set_title(f"Map game {team_home} - {team_away} {team_season}")
ax.legend(loc='lower center', bbox_to_anchor=(0.5, -0.2), ncol=2)

# Save the figure and show it
plt.savefig("Player_Shot_Chart.png", dpi=300, bbox_inches='tight')
plt.show()
```



DATA VISUALIZATION

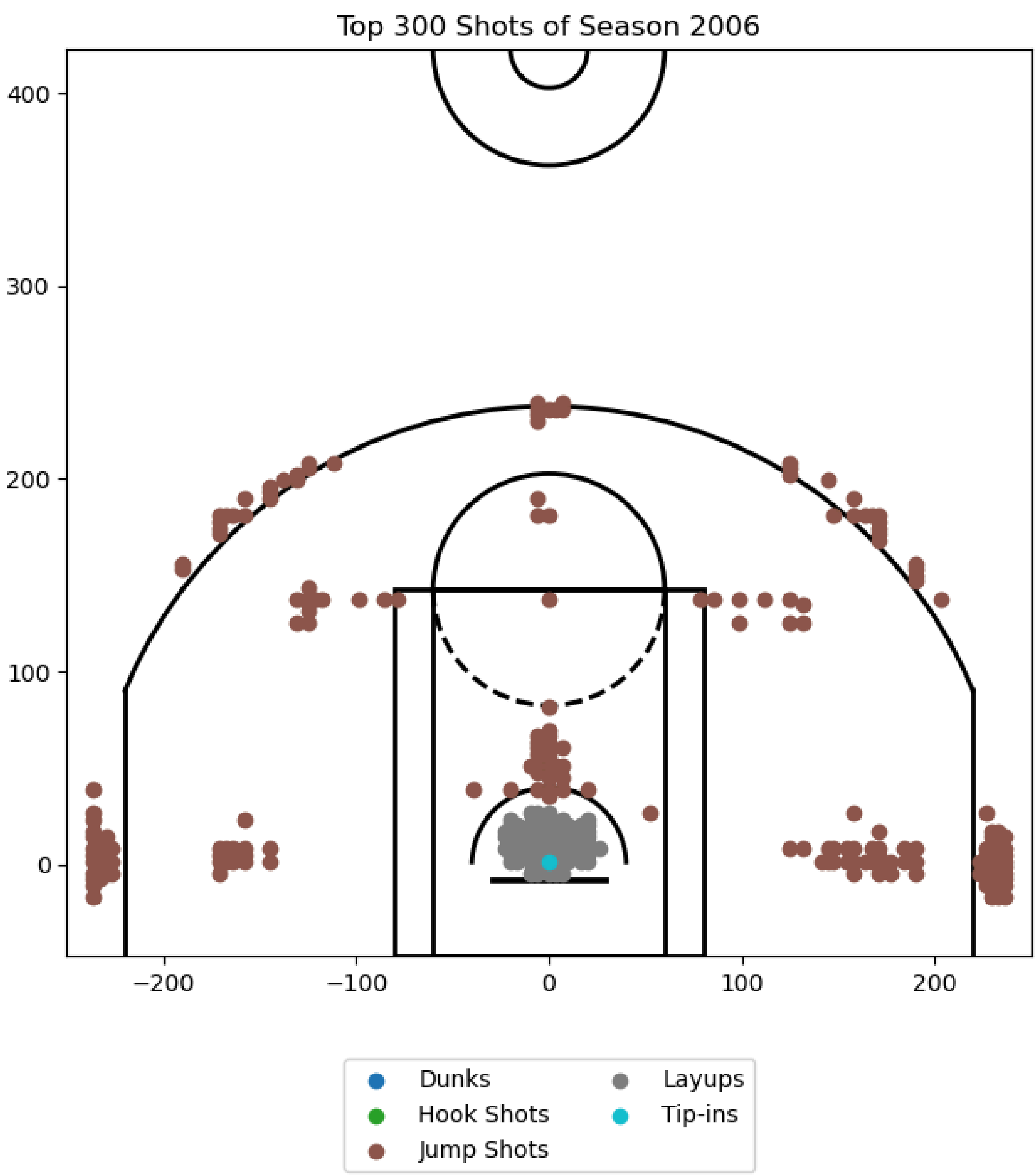
% success rate of shots from every BASIC_ZONE

In [14]:
```
# Put back the loc_x and y in feet (for technical reasons)
season_1['LOC_X_rounded'] = season_1['LOC_X'].round(decimals=1) / 0.3048
season_1['LOC_Y_rounded'] = season_1['LOC_Y'].round(decimals=1) / 0.3048
grouped_shots = season_1.groupby(['LOC_X_rounded', 'LOC_Y_rounded', 'SHOT

top_shots = grouped_shots.sort_values('Count', ascending=False).head(300)
```

In [15]:
```
# Define a color map for different action types
colors = plt.cm.get_cmap('tab10', len(top_shots['SHOT_CATEGORY'].unique()

# Create a figure and a single set of axes
fig, ax = plt.subplots(figsize=(10, 7))

# Draw the court on the main plot
draw_court(ax)
```

```python
# Plot each group with a different color in the main subplot
handles = []
labels = []
for idx, (action_type, group) in enumerate(top_shots.groupby('SHOT_CATEGO
    color = colors(idx)
    scatter = ax.scatter(group['LOC_X_rounded'] * 10, group['LOC_Y_rounde
    handles.append(scatter)
    labels.append(action_type)

# Set the title
ax.set_title(f"Top 300 Shots of Season {season_1['SEASON_1'].iloc[0]}")

# Create the legend within the same plot
ax.legend(handles=handles, labels=labels, loc='upper center', bbox_to_anc

# Show the plot
plt.show()
```

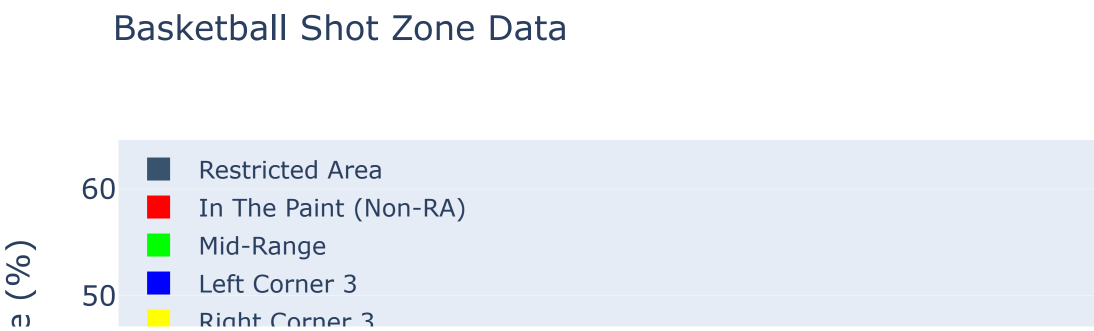
```
/var/folders/pw/1cm49_4j4bn00208sw3x8m2m0000gn/T/ipykernel_46451/131387418
5.py:2: MatplotlibDeprecationWarning: The get_cmap function was deprecated
in Matplotlib 3.7 and will be removed two minor releases later. Use ``matp
lotlib.colormaps[name]`` or ``matplotlib.colormaps.get_cmap(obj)`` instea
d.
  colors = plt.cm.get_cmap('tab10', len(top_shots['SHOT_CATEGORY'].unique
()))
```



In [16]:
```
def zone_average(season):
    # Calculate the total number of shots for each BASIC_ZONE
    total_shots = season.groupby("BASIC_ZONE").size()

    # Calculate the percentage of shots made for each BASIC_ZONE
    percentage_shots = season.groupby("BASIC_ZONE")["SHOT_MADE"].mean() *

    # Define colors for each zone
    zone_colors = {
        'Restricted Area': 'rgb(55, 83, 109)',
        'In The Paint (Non-RA)': 'rgb(255, 0, 0)',
        'Mid-Range': 'rgb(0, 255, 0)',
        'Left Corner 3': 'rgb(0, 0, 255)',
        'Right Corner 3': 'rgb(255, 255, 0)',
        'Above the Break 3': 'rgb(255, 0, 255)',
        'Backcourt' : 'rgb(0,0,0)'
    }

    # Create the bar plot
    fig = go.Figure()
```

```python
    for zone, color in zone_colors.items():
        shots_in_zone = season[data_shots["BASIC_ZONE"] == zone]
        fig.add_trace(go.Bar(
            x=[total_shots[zone]],
            y=[percentage_shots[zone]],
            name=zone,
            text=[zone],  # Show BASIC_ZONE as hover text
            hoverinfo='text+y',  # Show BASIC_ZONE and total number of sh
            marker_color=color
        ))

    # Customize the layout
    fig.update_layout(
        title='Basketball Shot Zone Data',
        xaxis=dict(
            title='Total Number of Shots',
            titlefont_size=16,
            tickfont_size=14,
        ),
        yaxis=dict(
            title='Percentage of Shots Made (%)',
            titlefont_size=16,
            tickfont_size=14,
        ),
        legend=dict(
            x=0,
            y=1.0,
            bgcolor='rgba(255, 255, 255, 0)',
            bordercolor='rgba(255, 255, 255, 0)'
        ),
        hovermode='closest',  # Show hover information for the nearest da
        bargap=0,  # gap between bars of adjacent location coordinates
        bargroupgap=0.1  # gap between bars of the same location coordina
    )

    # Show the plot
    fig.show()
```

In [17]:
```python
zone_average(data_shots)
```

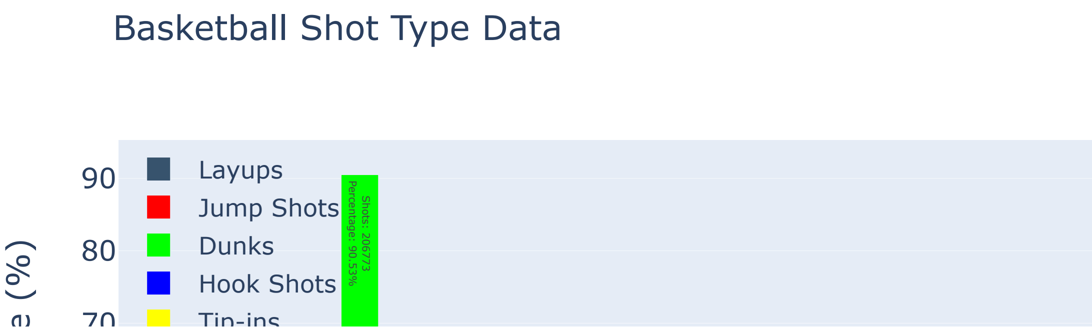


In [18]:
```
def shot_types_average(season):
    # Define colors for each shot category
    category_colors = {
        'Layups': 'rgb(55, 83, 109)',
        'Jump Shots': 'rgb(255, 0, 0)',
        'Dunks': 'rgb(0, 255, 0)',
        'Hook Shots': 'rgb(0, 0, 255)',
        'Tip-ins': 'rgb(255,255,0)',
        'No_Shot': 'rgb(0,0,0)'
        # Add more categories and colors as needed
    }

    # Filter data for the season and calculate shot made percentage per a
    shots_per_type = season.groupby("SHOT_CATEGORY").size()
    shots_per_type_percentage = season.groupby("SHOT_CATEGORY")["SHOT_MAD

    # Create the bar plot
    fig = go.Figure()
    for category, color in category_colors.items():
        if category in shots_per_type.index:
            fig.add_trace(go.Bar(
                x=[shots_per_type[category]],
                y=[shots_per_type_percentage[category]],
                name=category,
                text=f'Shots: {shots_per_type[category]}<br>Percentage: {
                hoverinfo='text',  # Show SHOT_CATEGORY, number of shots,
                marker_color=color
            ))
```

```python
    # Customize the layout
    fig.update_layout(
        title='Basketball Shot Type Data',
        xaxis=dict(
            title='Total Number of Shots',
            titlefont_size=16,
            tickfont_size=14,
        ),
        yaxis=dict(
            title='Percentage of Shots Made (%)',
            titlefont_size=16,
            tickfont_size=14,
        ),
        legend=dict(
            x=0,
            y=1.0,
            bgcolor='rgba(255, 255, 255, 0)',
            bordercolor='rgba(255, 255, 255, 0)'
        ),
        bargap=0.15,  # gap between bars of adjacent location coordinates
        bargroupgap=0.1  # gap between bars of the same location coordina
    )

    # Show the plot
    fig.show()
```

In [19]:

```python
# Call the function to display the graph
shot_types_average(data_shots)
```



HYPOTESIS TESTING

Step 5: Hypothesis Testing for Coefficients Hypotheses:

AWAY TEAM vs HOME TEAM shot success

Hypothesis: There is no significant difference in shooting percentages between home and away games for NBA players.

Null Hypothesis (H0): The mean shooting percentage for home games is equal to the mean shooting percentage for away games. Alternative Hypothesis (H1): The mean shooting percentage for home games is not equal to the mean shooting percentage for away games.

In [20]:
```
# Create a dictionary containing the Team ammbreaviations and the corresp
team_abbreviations = {
    "ATL": "Atlanta Hawks",
    "BOS": "Boston Celtics",
    "BKN": "Brooklyn Nets",
    "CHA": "Charlotte Hornets",
    "CHI": "Chicago Bulls",
    "CLE": "Cleveland Cavaliers",
    "DAL": "Dallas Mavericks",
    "DEN": "Denver Nuggets",
```

```
    "DET": "Detroit Pistons",
    "GSW": "Golden State Warriors",
    "HOU": "Houston Rockets",
    "IND": "Indiana Pacers",
    "LAC": "Los Angeles Clippers",
    "LAL": "Los Angeles Lakers",
    "MEM": "Memphis Grizzlies",
    "MIA": "Miami Heat",
    "MIL": "Milwaukee Bucks",
    "MIN": "Minnesota Timberwolves",
    "NJN": "New Jersey Nets",
    "NOH": "New Orleans Hornets",
    "NOP": "New Orleans Pelicans",
    "NOK": "New Orleans/Oklahoma City Hornets",
    "NYK": "New York Knicks",
    "OKC": "Oklahoma City Thunder",
    "ORL": "Orlando Magic",
    "PHI": "Philadelphia 76ers",
    "PHX": "Phoenix Suns",
    "POR": "Portland Trail Blazers",
    "SAC": "Sacramento Kings",
    "SAS": "San Antonio Spurs",
    "SEA": "Seattle SuperSonics",
    "TOR": "Toronto Raptors",
    "UTA": "Utah Jazz",
    "WAS": "Washington Wizards"
}

team_abbreviations = dict(sorted(team_abbreviations.items()))
```

In [21]:
```
def is_home_shot(row):
    return row['TEAM_NAME'] == team_abbreviations[row['HOME_TEAM']]
```

In [22]:
```
# Create a new column indicating whether the shot was made by the home te
season_1['IS_HOME_SHOT'] = season_1.apply(is_home_shot, axis=1)

# Divide the values of the shots from Home and Away teams
home_shots = season_1[season_1['IS_HOME_SHOT']]['SHOT_MADE']
away_shots = season_1[~season_1['IS_HOME_SHOT']]['SHOT_MADE']

# Calculate the Mean of the two populations
prop_home = home_shots.mean()
prop_away = away_shots.mean()

# Calculate pooled proportion
pooled_prop = (home_shots.sum() + away_shots.sum() + 0.5) / (len(home_sho

# Calculate standard error
se = np.sqrt(pooled_prop * (1 - pooled_prop) * (1 / len(home_shots) + 1 /

# Calculate z-statistic
z_stat = (prop_home - prop_away) / se

# Two-tailed test, so multiply p-value by 2
p_value = 2 * (1 - stats.norm.cdf(np.abs(z_stat)))

# Print the results
print(f"Z-Statistic: {z_stat}")
print(f"P-Value: {p_value}")
```

```
# Interpretation
alpha = 0.05
if p_value < alpha:
    print("Reject the null hypothesis. There is a significant difference
else:
    print("Fail to reject the null hypothesis. There is no significant di
```

```
Z-Statistic: 6.209925857917311
P-Value: 5.300959671217242e-10
Reject the null hypothesis. There is a significant difference in shooting
proportions between home and away teams.
```

Hypotesis: final quarters goal average vs first quarter goal average

Null Hypothesis ($H0$): There is no difference between the average points scored in the 1st quarter and the last quarter of NBA games.

Alternative Hypothesis ($H1$): There is a significant difference between the average points scored in the 1st quarter and the last quarter of NBA games.

In [23]:
```
# Group by 'GAME_ID' and 'QUARTER', then calculate the mean points for ea
season_1 = season_1.groupby(['GAME_ID', 'QUARTER'])['POINTS'].mean().rese

# Filter for the first and last quarters
first_quarter = season_1[season_1['QUARTER'] == 1][['GAME_ID', 'POINTS']]
last_quarter = season_1[season_1['QUARTER'] == 4][['GAME_ID', 'POINTS']]

# Rename columns to distinguish between the two quarters
first_quarter = first_quarter.rename(columns={'POINTS': 'points_first'})
last_quarter = last_quarter.rename(columns={'POINTS': 'points_last'})
```

In [24]:
```
# Merge the data on 'GAME_ID' to get pairs of points for each game
merged_data = pd.merge(first_quarter, last_quarter, on='GAME_ID')

# Calculate the differences
merged_data['diff'] = merged_data['points_first'] - merged_data['points_l

mean_diff = merged_data['points_first'] - merged_data['points_last']

# Calculate the standard deviation of the mean difference
std_diff = mean_diff.std()

# Calculate the standard error of the mean difference
n = len(merged_data)
se_diff = std_diff / (n ** 0.5)

# Calculate the t-statistic
t_stat = mean_diff.mean() / se_diff

# Degrees of freedom
df = n - 1

# Calculate the p-value (two-tailed test)
p_value = 2 * (1 - stats.t.cdf(abs(t_stat), df))

# Print the results
print(f'T-statistic: {t_stat}')
print(f'P-value: {p_value}')
```

```python
# Interpretation
if p_value < alpha:
    print("Reject the null hypothesis: There is a significant difference
else:
    print("Fail to reject the null hypothesis: There is no significant di
```

```
T-statistic: 4.518947420007024
P-value: 6.813627604573824e-06
Reject the null hypothesis: There is a significant difference between the average points scored in the 1st quarter and the last quarter.
```

Hypotesis: angle vs center shots

Null Hypothesis (H0): There is no difference in the shooting success rates between center shots and angle shots.

Alternative Hypothesis (H1): There is a difference in the shooting success rates between center shots and angle shots.

In [25]:
```python
# Divide center and side shots in two Dataframes
season_1 = season_2
center_shots = season_1[((season_1['ANGLE_FROM_CENTER_DEGREES']) > (60))
side_shots = season_1[((season_1['ANGLE_FROM_CENTER_DEGREES']) <= (60))][
side_shots = pd.concat([side_shots, season_1[((season_1['ANGLE_FROM_CENTE

# Rename columns
center_shots = center_shots.rename(columns={'POINTS': 'center_points'})
side_shots = side_shots.rename(columns={'POINTS': 'side_points'})
```

In [26]:
```python
# Merge the data on 'GAME_ID' to get pairs of points for each game
merged_data = pd.merge(center_shots, side_shots, on='GAME_ID')

# Calculate the differences
merged_data['diff'] = merged_data['center_points'] - merged_data['side_po

# Calculate the t-statistic and the p-value
t_stat, p_value = stats.ttest_rel(merged_data['center_points'], merged_da

# Print the results
print(f'T-statistic: {t_stat}')
print(f'P-value: {p_value}')

# Interpretation
if p_value < alpha:
    print("Reject the null hypothesis: There is a significant difference
else:
    print("Fail to reject the null hypothesis: There is no significant di
```

```
T-statistic: 334.1861734423591
P-value: 0.0
Reject the null hypothesis: There is a significant difference between the average points scored in the 1st quarter and the last quarter.
```

LINEAR REGRESSION

TRY OUT FOR MULTI LINEAR REGRESSION

In [27]:
```
# LINEAR STANDARD REGRESSION, WITH DISTANCE TO BASKET AS INDEPENDENT VARI
"""
# Define intervals
intervals = [(0, 8), (9, 16), (17, 24), (25, 32), (33, 40), (41, 48), (49

# Use shot distance and points
shot_distance = data_shots['SHOT_DISTANCE'].values
shot_points = data_shots['POINTS'].values

# Calculate the average shot points for each interval
avg_shot_points_list = []
interval_mid_points = []

for interval in intervals:
    start, end = interval
    mask = (shot_distance >= start) & (shot_distance < end)
    avg_shot_points = np.mean(shot_points[mask])
    avg_shot_points_list.append(avg_shot_points)
    interval_mid_points.append((start + end) / 2)

# Calculate the slope (m) and intercept (c) for the linear regression lin
n = len(interval_mid_points)
m = (n * np.sum(np.array(interval_mid_points) * np.array(avg_shot_points_
c = (np.sum(avg_shot_points_list) - m * np.sum(interval_mid_points)) / n

# Generate regression line
reg_line = m * np.array(interval_mid_points) + c

# Plot the average points for each interval and the regression line
plt.figure(figsize=(10, 6))
plt.scatter(interval_mid_points, avg_shot_points_list, color='blue', alph
plt.plot(interval_mid_points, reg_line, color='red')
plt.title('Average Points Scored vs Shot Distance')
plt.xlabel('Shot Distance (feet)')
plt.ylabel('Average Points Scored')
plt.grid(True)
plt.show()
"""
```

Out[27]: "\n# Define intervals\nintervals = [(0, 8), (9, 16), (17, 24), (25, 32), (33, 40), (41, 48), (49,56), (57,64), (65,72), (75,80), (81,88)]\n\n# Use shot distance and points\nshot_distance = data_shots['SHOT_DISTANCE'].values\nshot_points = data_shots['POINTS'].values\n\n# Calculate the average shot points for each interval\navg_shot_points_list = []\ninterval_mid_points = []\n\nfor interval in intervals:\n   start, end = interval\n   mask = (shot_distance >= start) & (shot_distance < end)\n   avg_shot_points = np.mean(shot_points[mask])\n   avg_shot_points_list.append(avg_shot_points)\n   interval_mid_points.append((start + end) / 2)\n\n# Calculate the slope (m) and intercept (c) for the linear regression line\nn = len(interval_mid_points)\nm = (n * np.sum(np.array(interval_mid_points) * np.array(avg_shot_points_list)) - np.sum(interval_mid_points) * np.sum(avg_shot_points_list)) / (n * np.sum(np.array(interval_mid_points)**2) - np.sum(interval_mid_points)**2)\nc = (np.sum(avg_shot_points_list) - m * np.sum(interval_mid_points)) / n\n\n# Generate regression line\nreg_line = m * np.array(interval_mid_points) + c\n\n# Plot the average points for each interval and the regression line\nplt.figure(figsize=(10, 6))\nplt.scatter(interval_mid_points, avg_shot_points_list, color='blue', alpha=0.5)\nplt.plot(interval_mid_points, reg_line, color='red')\nplt.title('Average Points Scored vs Shot Distance')\nplt.xlabel('Shot Distance (feet)')\nplt.ylabel('Average Points Scored')\nplt.grid(True)\nplt.show()\n"

In [28]:
```
def linear_regression(X, Y):
    """
    Calculates the coefficients of linear regression using the least squa

    Arguments:
    X -- List of lists containing the feature values
    Y -- List containing the target variable values

    Returns:
    coefficients -- Tuple containing the coefficients of the linear regre
    """
    n_samples = len(X)
    n_features = len(X[0])  # Number of features

    # Step 1: Calculate the means of X and Y
    mean_X = [sum(X) / n_samples for X in zip(*X)]
    mean_Y = sum(Y) / n_samples

    # Step 2: Calculate the deviations and the products of the deviations
    deviations_X = [[X[i][j] - mean_X[j] for j in range(n_features)] for
    deviations_Y = [Y[i] - mean_Y for i in range(n_samples)]

    # Step 3: Calculate the sums of the products of the deviations
    sum_dev_XY = [sum(deviations_X[i][j] * deviations_Y[i] for i in range
    sum_dev_XX = [sum(deviations_X[i][j] * deviations_X[i][j] for i in ra

    # Step 4: Calculate the coefficients (slopes)
    m = [sum_dev_XY[j] / (sum_dev_XX[j] + 1e-8) if sum_dev_XX[j] != 0 els

    # Step 5: Calculate the intercept
    c = mean_Y - sum(m[j] * mean_X[j] for j in range(n_features))

    return m, c
```

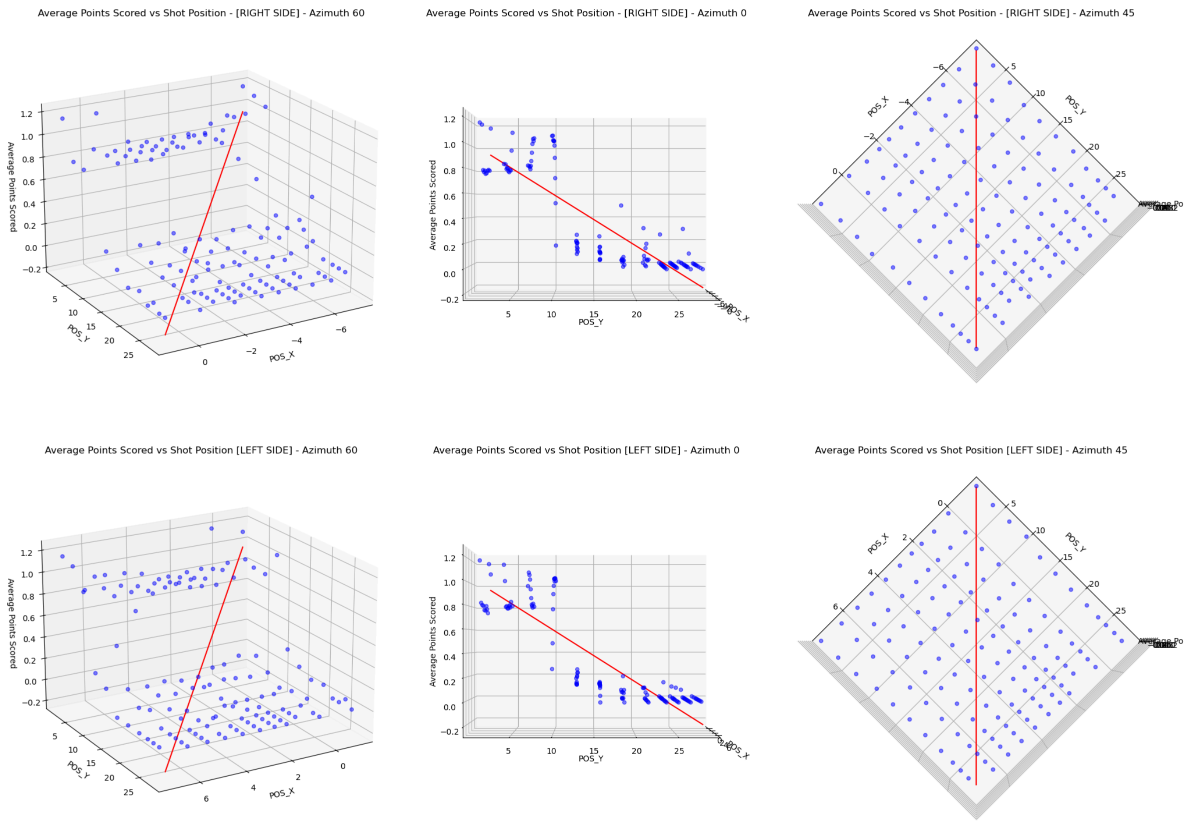
In [29]:
```
def gaussian_elimination(A, b):
    """
```

```
    Solve the linear system Ax = b using Gaussian elimination.

    Arguments:
    A -- List of lists containing the coefficient matrix
    b -- List containing the right-hand side

    Returns:
    x -- List containing the solution of the linear system
    """
    n = len(A)
    epsilon = 1e-8  # Regularization constant

    for k in range(n):
        # Find the maximum pivoting element
        max_row = k
        for i in range(k + 1, n):
            if abs(A[i][k]) > abs(A[max_row][k]):
                max_row = i

        # Swap rows
        A[k], A[max_row] = A[max_row], A[k]
        b[k], b[max_row] = b[max_row], b[k]

        # Eliminate elements below the pivot
        for i in range(k + 1, n):
            factor = A[i][k] / (A[k][k] + epsilon)
            for j in range(k, n):
                A[i][j] -= factor * A[k][j]
            b[i] -= factor * b[k]

    # Back substitution
    x = [0] * n
    for k in range(n - 1, -1, -1):
        x[k] = (b[k] - sum(A[k][j] * x[j] for j in range(k + 1, n))) / (A

    return x
```

In [30]:

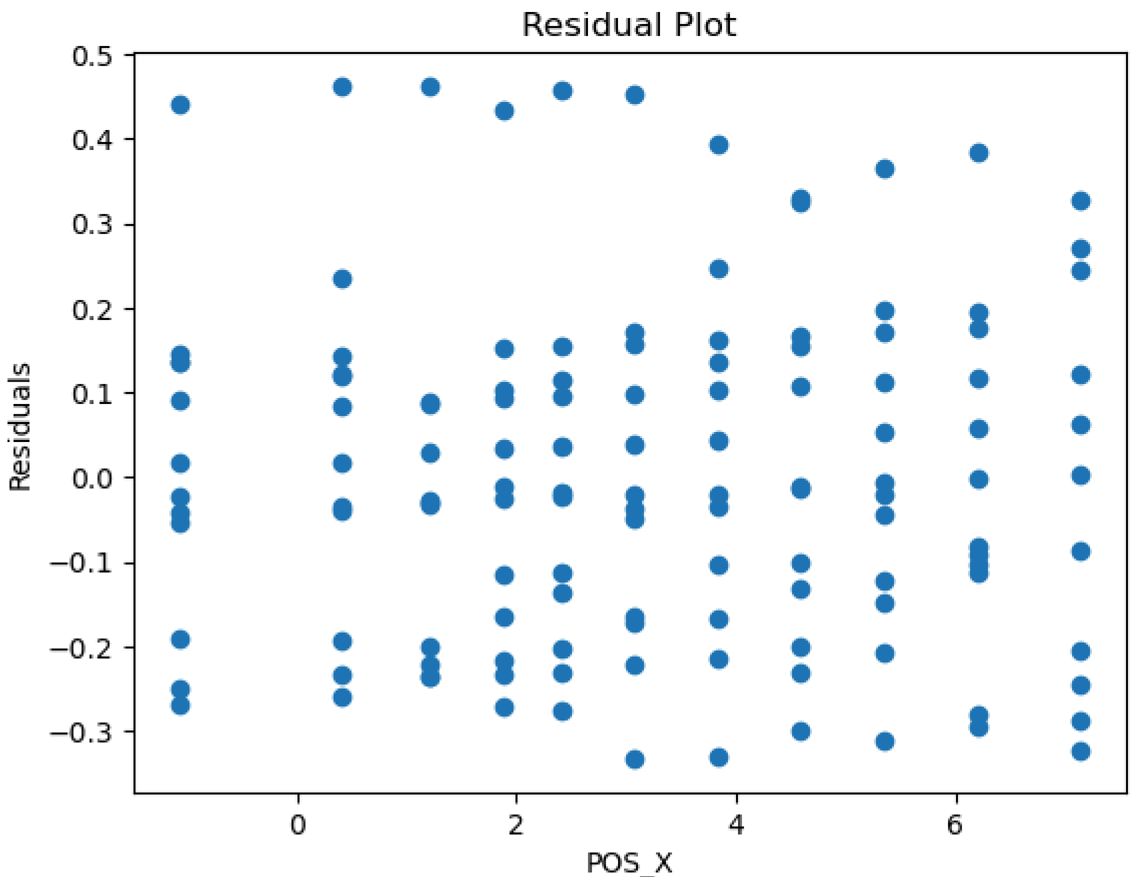
```
# Define intervals for POS_X for both plots
x_intervals_1 = [(-7.62, -6.678), (-6.678,-5.736), (-5.736, -4.974), (-4.
x_intervals_2 = [(-2.156,0), (0,0), (0, 0.812), (0.812, 1.624), (1.624, 2

# Define intervals for POS_Y for both plots
y_intervals = [(0,0), (0, 2.865), (2.865, 5.73), (5.73, 8.595), (8.595, 1

# Use POS_X, POS_Y, and POINTS
pos_x = data_shots['LOC_X']
pos_y = data_shots['LOC_Y']
shot_points = data_shots['POINTS']

# Create a figure
fig = plt.figure(figsize=(20, 16))

titles = ['Average Points Scored vs Shot Position - [RIGHT SIDE]', 'Avera
# Define the azimuth angles for the three points of view
azimuths = [60, 0, 45]
elevations = [20, 0, 90]

for i, x_intervals in enumerate([x_intervals_1, x_intervals_2]):
    avg_shot_points_list = []
    interval_mid_points_x = []
```

```
    interval_mid_points_y = []

    for x_interval in x_intervals:
        for y_interval in y_intervals:
            x_start, x_end = x_interval
            y_start, y_end = y_interval

            x_mask = (pos_x >= x_start) & (pos_x < x_end)
            y_mask = (pos_y >= y_start) & (pos_y < y_end)
            mask = x_mask & y_mask

            if np.any(mask):
                avg_shot_points = np.mean(shot_points[mask])
            else:
                avg_shot_points = np.nan  # Handle empty intervals
            avg_shot_points_list.append(avg_shot_points)

            interval_mid_points_x.append((x_start + x_end) / 2)
            interval_mid_points_y.append((y_start + y_end) / 2)

    # Remove nan values from avg_shot_points_list and corresponding mid p
    valid_mask = ~np.isnan(avg_shot_points_list)
    interval_mid_points_x = np.array(interval_mid_points_x)[valid_mask]
    interval_mid_points_y = np.array(interval_mid_points_y)[valid_mask]
    avg_shot_points_list = np.array(avg_shot_points_list)[valid_mask]

    # Perform linear regression from scratch
    X = [[x, y, 1] for x, y in zip(interval_mid_points_x, interval_mid_po
    Y = avg_shot_points_list.tolist()

    # Solve for coefficients using the normal equation
    coefficients = linear_regression(X, Y)
    m, c = coefficients

    # Generate points for the regression line
    line_x = np.linspace(interval_mid_points_x.min(), interval_mid_points
    line_y = np.linspace(interval_mid_points_y.min(), interval_mid_points
    line_z = m[0] * line_x + m[1] * line_y + c


    # Plot the average points for each interval and the regression line
    for j, (azim, el) in enumerate(zip(azimuths, elevations)):
        ax = fig.add_subplot(2, 3, i * 3 + j + 1, projection='3d')
        ax.scatter(interval_mid_points_x, interval_mid_points_y, avg_shot
        ax.plot(line_x, line_y, line_z, color='red')

        ax.set_title(f'{titles[i]} - Azimuth {azim}')
        ax.set_xlabel('POS_X')
        ax.set_ylabel('POS_Y')
        ax.set_zlabel('Average Points Scored')

        ax.view_init(elev=el, azim=azim)

plt.tight_layout()
plt.show()
```



In [31]:

```
residuals = avg_shot_points_list - (m[0] * interval_mid_points_x + m[1] *
plt.scatter(interval_mid_points_x, residuals)
plt.xlabel('POS_X')
plt.ylabel('Residuals')
plt.title('Residual Plot')
plt.show()
```



In [32]:
```
from sklearn.metrics import r2_score
r2 = r2_score(avg_shot_points_list, m[0] * interval_mid_points_x + m[1] *
print(f'R-squared: {r2}')
```

```
R-squared: 0.7636855859985298
```

The residuals are randomly scattered around zero, and they don't follow any pattern. Thus the model is valid

R^2 is high enough to tell that 76% of the predicted variables is affected by the independent variable

In [33]:
```
# Calculate average points for each shot category
total_points_per_category = data_shots.groupby('SHOT_CATEGORY')['POINTS']
count_per_category = data_shots['SHOT_CATEGORY'].value_counts().to_dict()
avg_points_per_category = {category: total_points_per_category[category]

# Calculate average points for each basic zone
total_points_per_zone = data_shots.groupby('BASIC_ZONE')['POINTS'].sum().
count_per_zone = data_shots['BASIC_ZONE'].value_counts().to_dict()
avg_points_per_zone = {zone: total_points_per_zone[zone] / count_per_zone

# Calculate average points for each quarter
#total_points_per_quarter = data_shots.groupby('QUARTER')['POINTS'].sum()
#count_per_quarter = data_shots['QUARTER'].value_counts().to_dict()
#avg_points_per_quarter = {quarter: total_points_per_quarter[quarter] / c

# Replace one-hot encoding with average points
def encode_shot_category(category):
    return [avg_points_per_category.get(category, 0)]

def encode_basic_zone(zone):
    return [avg_points_per_zone.get(zone, 0)]

#def encode_quarter(quarter):
    #return [avg_points_per_quarter.get(quarter, 0)]
```

In [34]:
```
def linear_regression_2(X, Y):
    """
    Calculate the coefficients of linear regression using the least squar

    Arguments:
    X -- List of lists containing the feature values
    Y -- List containing the target variable values

    Returns:
    coefficients -- List containing the coefficients of linear regression
    """

    #Convert the list of lists X and the list Y to numpy arrays for easie
    X = np.array(X)
    Y = np.array(Y)

    # Compute the dot product of the transpose of X with X.
    # This results in a square matrix that is the sum of the outer produc
    X_T_X = np.dot(X.T, X)

    # Compute the dot product of the transpose of X with Y.
```

```
    # This results in a vector where each element is the sum of the produ
    X_T_Y = np.dot(X.T, Y)

    # Solve the linear system of equations X_T_X * coefficients = X_T_Y t
    # This step uses numpy's linear algebra solver to find the vector of
    # It uses LU decomposition and forward/backward substitution
    coefficients = np.linalg.solve(X_T_X, X_T_Y)


    return coefficients
```

MODEL SELECTION:

In [35]:
```
def calculate_r_squared(X, Y, coefficients):
    # Predict the target values using the coefficients
    y_pred = np.dot(X, coefficients)

    # Calculate the total sum of squares (variance of Y)
    ss_total = np.sum((Y - np.mean(Y))**2)

    # Calculate the residual sum of squares (difference between actual an
    ss_residual = np.sum((Y - y_pred)**2)

    # Calculate R-squared as the proportion of variance explained by the
    r_squared = 1 - (ss_residual / ss_total)
    return r_squared
```

In [36]:
```
# Function to calculate Adjusted R-squared
def calculate_adjusted_r_squared(X, Y, coefficients):
    # Number of observations
    n = len(Y)

    # Number of predictors (excluding the intercept term)
    # 'X.shape' returns the dimensions of the array X as a tuple (n_sampl
    # '[1]' gives the number of columns in X, which corresponds to the nu
    # the last column of X is the intercept term (a column of ones) thus

    k = X.shape[1] - 1

    #from the previous func
    r_squared = calculate_r_squared(X, Y, coefficients)


    adjusted_r_squared = 1 - ((1 - r_squared) * (n - 1) / (n - k - 1))
    return adjusted_r_squared
```

In [37]:
```
# Backward Elimination function
def backward_elimination(X, Y, significance_level=0.05):
    """
    OLS estimates the coefficients  by minimizing the sum of the squared
    Backward elimination to select the most significant features for a li

    Arguments:
    X -- 2D numpy array of shape (n_samples, n_features), where each row
    Y -- 1D numpy array of shape (n_samples,), containing the target vari

    Returns:
    X -- The dataset with only the remaining significant predictors.
    """
```

```
    # Number of variables (features) in the dataset
    num_vars = X.shape[1]

    for i in range(num_vars):
        # Fit the Ordinary Least Squares (OLS) model
        # 'sm.OLS' creates an OLS model object with Y as the dependent va
        # '.fit()' estimates the coefficients that minimize the sum of sq
        # basically it just does the same thing of linear_regression_2, b
        regressor_OLS = sm.OLS(Y, X).fit()

        # Find the maximum p-value among the predictors
        max_p_value = max(regressor_OLS.pvalues).astype(float)

        # If the maximum p-value is greater than the significance level,
        if max_p_value > significance_level:
            for j in range(num_vars - i):
                if regressor_OLS.pvalues[j].astype(float) == max_p_value:
                    # Remove the column (predictor) with the highest p-va
                    # The '1' indicates that the deletion is along the co
                    X = np.delete(X, j, 1)

        # Print the summary of the model (like coefficients, p-values, SE
        regressor_OLS.summary()

    # Return the dataset with the remaining predictors
    return X
```

In [38]:
```
# Forward Selection function
def forward_selection(X, Y, significance_level=0.05):
    selected_features = []
    remaining_features = list(range(X.shape[1]))
    while remaining_features:
        remaining_p_values = []
        for feature in remaining_features:
            selected_features.append(feature)
            X_selected = X[:, selected_features]
            regressor_OLS = sm.OLS(Y, X_selected).fit()
            p_value = regressor_OLS.pvalues[-1]
            remaining_p_values.append(p_value)
            selected_features.pop()
        min_p_value = min(remaining_p_values)
        if min_p_value < significance_level:
            min_p_value_index = remaining_p_values.index(min_p_value)
            selected_features.append(remaining_features[min_p_value_index
            remaining_features.pop(min_p_value_index)
        else:
            break
    X_selected = X[:, selected_features]
    return X_selected
```

CONFIDENCE INTERVAL for the multilinear regression

In [39]:
```
def calculate_confidence_intervals_z(X, Y, coefficients, alpha=0.05):
    """
    Calculate confidence intervals for regression coefficients using the

    Arguments:
    X -- 2D numpy array of shape (n_samples, n_features), the design matr
```

```python
    Y -- 1D numpy array of shape (n_samples,), the target variable.
    coefficients -- 1D numpy array of shape (n_features,), the estimated
    alpha -- significance level for the confidence intervals (default is

    Returns:
    lower_bounds -- 1D numpy array of shape (n_features,), the lower boun
    upper_bounds -- 1D numpy array of shape (n_features,), the upper boun
    """

    # Convert X and Y to numpy arrays (if they aren't already)
    X = np.array(X)
    Y = np.array(Y)

    # Calculate predictions using the regression coefficients
    predictions = np.dot(X, coefficients)

    # Calculate residuals (differences between actual and predicted value
    residuals = Y - predictions

    # Sum of squared residuals
    residual_sum_of_squares = np.sum(residuals**2)

    # Degrees of freedom (number of observations minus the number of pred
    degrees_of_freedom = X.shape[0] - X.shape[1]

    # Variance of the residuals (residual sum of squares divided by degre
    residual_variance = residual_sum_of_squares / degrees_of_freedom

    # Calculate the variance-covariance matrix of the coefficients
    # X.T is the transpose of X
    # np.dot(X.T, X) is the matrix product of X transpose and X
    # np.linalg.inv() computes the inverse of the matrix
    XtX_inv = np.linalg.inv(np.dot(X.T, X))

    # The variance of the coefficients is the residual variance multiplie
    coefficient_variance = residual_variance * XtX_inv

    # The standard errors of the coefficients are the square roots of the
    standard_errors = np.sqrt(np.diag(coefficient_variance))

    # Calculate the z critical value for the given alpha level (e.g., 1.9
    z_critical = norm.ppf(1 - alpha/2)

    # Calculate the lower and upper bounds of the confidence intervals
    lower_bounds = coefficients - z_critical * standard_errors
    upper_bounds = coefficients + z_critical * standard_errors

    # Return the lower and upper bounds as a tuple
    return lower_bounds, upper_bounds
```

In [40]:

```python
def calculate_expected_average(x, y, shot_cat, bas_zone, coefficients):
    # Create the feature vector by combining:
    # - x and y coordinates
    # - Encoded shot category
    # - Encoded basic zone
    # - A constant term for the intercept (bias term)

    # Encode the shot category into a feature vector
    shot_cat_features = encode_shot_category(shot_cat)
```

```
    # Encode the basic zone into a feature vector
    bas_zone_features = encode_basic_zone(bas_zone)

    # Combine all features into a single list: x, y, encoded shot categor
    features = [x, y] + shot_cat_features + bas_zone_features + [1]

    # Calculate the expected average using the linear combination of coef
    expected_avg = sum(coef * feat for coef, feat in zip(coefficients, fe

    # Ensure the expected average is not negative (xP averages that go on
    return expected_avg if expected_avg > 0 else 0
```

"FINAL" MULTILINEAR REGRESSION

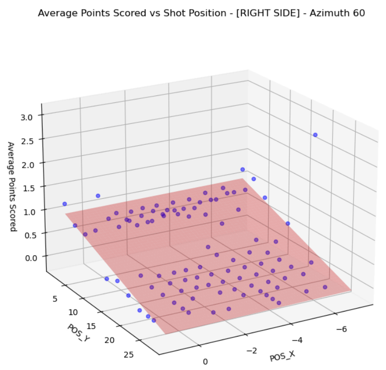
In [41]:
```
#we're defining intervals for the X and Y positions on the basketball cou
#These intervals will help us segment the court into smaller regions for

#x_i_1 represents the left side of the court in respect to the basket, x_
x_intervals_1 = [(-7.62, -6.678), (-6.678,-5.736), (-5.736, -4.974), (-4.
x_intervals_2 = [(-2.156,0), (0,0), (0, 0.812), (0.812, 1.624), (1.624, 2

#this interval must necessary be the same dimension as those of x
y_intervals = [(0,0), (0, 2.865), (2.865, 5.73), (5.73, 8.595), (8.595, 1

# Use POS_X, POS_Y, SHOT_CATEGORY, and POINTS
pos_x = season_1['LOC_X']
pos_y = season_1['LOC_Y']
shot_category = season_1['SHOT_CATEGORY']
basic_zone = season_1['BASIC_ZONE']
shot_points = season_1['POINTS']
#angle_shot = season_1['ANGLE']
#quarter_shot = season_1['QUARTER']

# Create a list of feature names
feature_names = ['POS_X', 'POS_Y', 'SHOT_CATEGORY', 'BASIC_ZONE', 'INTERC

# Create a figure
fig = plt.figure(figsize=(20, 16))

titles = ['Average Points Scored vs Shot Position - [RIGHT SIDE]', 'Avera
# Define the azimuth angles for the three points of view
angles = [60, 0, 45]
elevations = [20, 0, 90]

#This is the beginning of a loop where we're iterating over the two sets
#We'll generate plots for each set of intervals, representing different r
for i, x_intervals in enumerate([x_intervals_1, x_intervals_2]):
    avg_shot_points_list = []
    interval_mid_points_x = []
    interval_mid_points_y = []
    expected_avg_list = []  # Initialize a list to store expected average

    for x_interval in x_intervals:
        for y_interval in y_intervals:
            #We define the start and end points for the current X and Y i
            #to filter shots falling within these intervals.
            x_start, x_end = x_interval
            y_start, y_end = y_interval
```

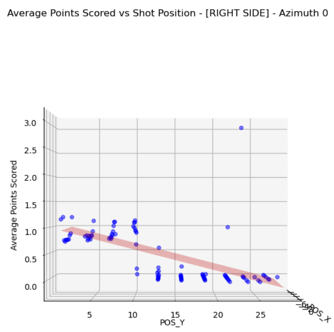
```
            x_mask = (pos_x >= x_start) & (pos_x < x_end)
            y_mask = (pos_y >= y_start) & (pos_y < y_end)
            mask = x_mask & y_mask

            #We calculate the average points scored for shots falling wit
            if np.any(mask):
                avg_shot_points = np.mean(shot_points[mask])
            else:
                avg_shot_points = np.nan  # Handle empty intervals

            #We append the average shot points and the mid-points of the
            avg_shot_points_list.append(avg_shot_points)
            interval_mid_points_x.append((x_start + x_end) / 2)
            interval_mid_points_y.append((y_start + y_end) / 2)

    #We create a boolean mask to filter out NaN values from avg_shot_poin
    #and use it to filter interval_mid_points_x and interval_mid_points_y
    valid_mask = ~np.isnan(avg_shot_points_list)
    interval_mid_points_x = np.array(interval_mid_points_x)[valid_mask]
    interval_mid_points_y = np.array(interval_mid_points_y)[valid_mask]
    avg_shot_points_list = np.array(avg_shot_points_list)[valid_mask]

    # Perform linear regression from scratch
    #We create the feature matrix X for linear regression, including shot
    X = [[x, y] + encode_shot_category(shot_cat) + encode_basic_zone(bas_
    #We also create the target vector Y from avg_shot_points_list.
    Y = avg_shot_points_list.tolist()

    # Solve for coefficients using the normal equation
    coefficients = linear_regression_2(X, Y)

    r_squared_full = calculate_adjusted_r_squared(np.array(X), Y, coeffic


    print()


    # Backward Elimination
    X_backward = backward_elimination(np.array(X), np.array(Y))
    coefficients_backward = linear_regression_2(X_backward, Y)

    # Forward Selection
    X_forward = forward_selection(np.array(X), np.array(Y))
    coefficients_forward = linear_regression_2(X_forward, Y)


    # Calculate R-squared and Adjusted R-squared for each model
    r_squared_full = calculate_r_squared(np.array(X), Y, coefficients)
    adj_r_squared_full = calculate_adjusted_r_squared(np.array(X), Y, coe

    r_squared_backward = calculate_r_squared(X_backward, Y, coefficients_
    adj_r_squared_backward = calculate_adjusted_r_squared(X_backward, Y,

    r_squared_forward = calculate_r_squared(X_forward, Y, coefficients_fo
    adj_r_squared_forward = calculate_adjusted_r_squared(X_forward, Y, co

   # Full model
    print("Full Model:")
    for coef, name in zip(coefficients, feature_names):
        print(f"{name}: {coef:.4f}")
```

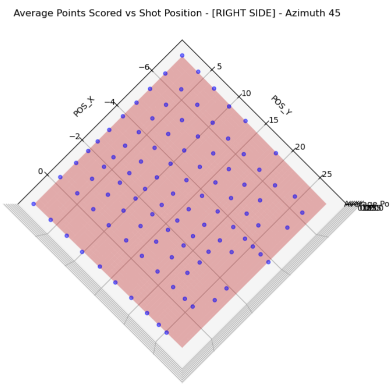
```
    print(f"R^2: {r_squared_full}")
    print(f"R^2 Adj: {adj_r_squared_full}")


    # Backward elimination model
    print("\nBackward Elimination Model:")
    for coef, name in zip(coefficients_backward, feature_names):
        if coef != 0:
            print(f"{name}: {coef:.4f}")
    print(f"R^2: {r_squared_backward}")
    print(f"R^2 Adj.: {adj_r_squared_backward}")

    # Forward selection model
    print("\nForward Selection Model:")
    for coef, name in zip(coefficients_forward, feature_names):
        if coef != 0:
            print(f"{name}: {coef:.4f}")
    print(f"R^2: {r_squared_forward}")
    print(f"R^2 Adj: {adj_r_squared_forward}")

    lower_bounds, upper_bounds = calculate_confidence_intervals_z(X, Y, c

    # Print the results
    print()
    print("Lower bounds of confidence intervals:", lower_bounds)
    print("Upper bounds of confidence intervals:", upper_bounds)

    # Number of coefficients
    num_coefficients = len(coefficients)

    #the first two coefficients are for m_x and m_y, the rest for the cat
    slopes = coefficients[:-1]
    c = coefficients[-1]

    # We generate points along the X and Y axes to plot the regression pl
    line_x = np.linspace(interval_mid_points_x.min(), interval_mid_points
    line_y = np.linspace(interval_mid_points_y.min(), interval_mid_points

    #We create a grid of points in the X-Y plane and initialize the Z val
    grid_x, grid_y = np.meshgrid(line_x, line_y)
    grid_z = np.zeros_like(grid_x)

    #Here, we calculate the Z values for each point in the grid. We itera
    #calculate the features for that point, and then calculate the corres
    for ix, x_val in enumerate(line_x):
        for iy, y_val in enumerate(line_y):
            features = [x_val, y_val]  # Initialize features with shot po
            for zone in basic_zone:
                features += encode_basic_zone(zone)  # Add one-hot encode
            for cat in shot_category:
                features += encode_shot_category(cat)
            #for angle in data_shots['ANGLE']:
                #features.append(angle)  # Add ANGLE feature
            #for quarter in data_shots['QUARTER']:
                #features += encode_quarter(quarter)  # Add one-hot encod

            grid_z[iy, ix] = sum(slope * feature for slope, feature in zi

    #We calculate the expected average points scored for each interval by
    for x, y, shot_cat, bas_zone in zip(interval_mid_points_x, interval_m
```

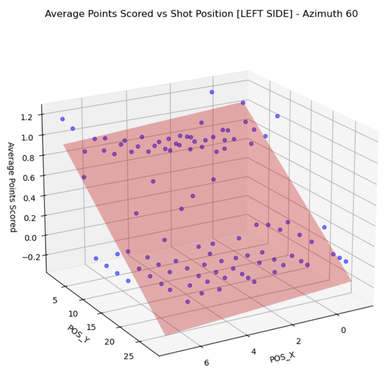
```
        # We calculate the features for each interval
        features = [x, y] + encode_shot_category(shot_cat) + encode_basic
        #. and use the obtained coefficients to calculate the expected av
        expected_avg = sum(coef * feat for coef, feat in zip(coefficients
        expected_avg_list.append(expected_avg)


    # Apply the function to each row in the DataFrame to calculate the ex
    # a new column xP_AVG is created to store these values.
    season_1['xP_AVG'] = season_1.apply(lambda row: calculate_expected_av


    # We iterate over azimuth and elevation angles to create three differ
    for j, (azim, el) in enumerate(zip(angles, elevations)):
        #or each view, we add a subplot to the figure
        ax = fig.add_subplot(2, 3, i * 3 + j + 1, projection='3d')
        #scatter the average points scored for each interval
        ax.scatter(interval_mid_points_x, interval_mid_points_y, avg_shot
        # plot the regression plane
        ax.plot_surface(grid_x, grid_y, grid_z, color='red', alpha=0.3, l

        ax.set_title(f'{titles[i]} - Azimuth {azim}')
        ax.set_xlabel('POS_X')
        ax.set_ylabel('POS_Y')
        ax.set_zlabel('Average Points Scored')
        #adjust the viewpoint by using the list I initialised at the begi
        ax.view_init(elev=el, azim=azim)

plt.tight_layout()
plt.show()
```

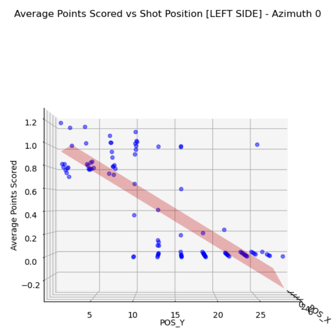
```
Full Model:
POS_X: -0.0035
POS_Y: -0.0402
SHOT_CATEGORY: -0.1906
BASIC_ZONE: 0.3380
INTERCEPT: 0.7994
R^2: 0.39184511404534106
R^2 Adj: 0.3665053271305636

Backward Elimination Model:
POS_X: -0.0404
POS_Y: 0.9649
R^2: 0.381272558063104
R^2 Adj.: 0.3750227859223273

Forward Selection Model:
POS_X: 0.2350
POS_Y: -0.0404
SHOT_CATEGORY: 0.7284
R^2: 0.3876273351839802
R^2 Adj: 0.3751299338612042

Lower bounds of confidence intervals: [-0.03778991 -0.05041614 -0.69164812
-0.19810588  0.24940284]
Upper bounds of confidence intervals: [ 0.03078067 -0.02998709  0.31053795
0.87403126  1.34943978]

Full Model:
POS_X: -0.0140
POS_Y: -0.0451
SHOT_CATEGORY: -0.3466
BASIC_ZONE: 0.1688
INTERCEPT: 1.2449
R^2: 0.6094616867001884
R^2 Adj: 0.5933570139867941

Backward Elimination Model:
POS_X: -0.0432
POS_Y: 1.0032
R^2: 0.5854344359566864
R^2 Adj.: 0.5812887803162532

Forward Selection Model:
POS_X: 1.0032
POS_Y: -0.0432
R^2: 0.5854344359566864
R^2 Adj: 0.5812887803162532

Lower bounds of confidence intervals: [-0.03729693 -0.0525737  -0.71021294
-0.22417846  0.87594701]
Upper bounds of confidence intervals: [ 0.00929431 -0.03767551  0.01697772
0.56171758  1.61381725]
```

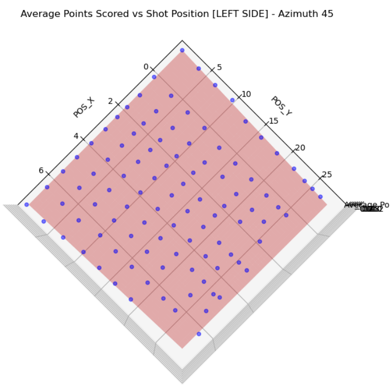












WHAT IF (I USED ANGLE AND QUARTER TOO):

Full Model: POS_X: -0.0043 POS_Y: -0.0403 ANGLE: 0.0004 SHOT_CATEGORY: -0.1994 BASIC_ZONE: 0.4131 QUARTER: 1.4328 INTERCEPT: -0.7420

Backward Elimination Model: POS_X: -0.0401 POS_Y: 0.9833

Forward Selection Model: POS_X: 0.9833 POS_Y: -0.0401

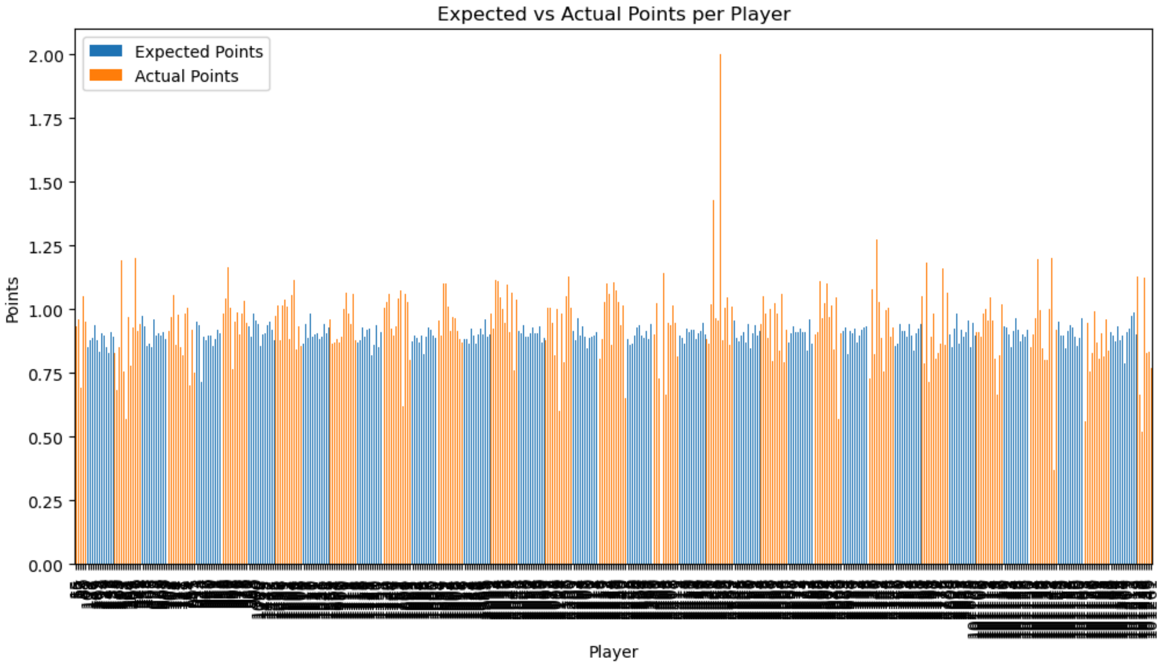
Lower bounds of confidence intervals: [-3.84929150e-02 -5.06124832e-02 -3.14093391e-04 -7.06341800e-01 -1.35933896e-01 -1.03139264e+00 -3.18075041e+00] Upper bounds of confidence intervals: [2.99134762e-02 -3.00368505e-02 1.10453798e-03 3.07635385e-01 9.62037985e-01 3.89695579e+00 1.69681026e+00]

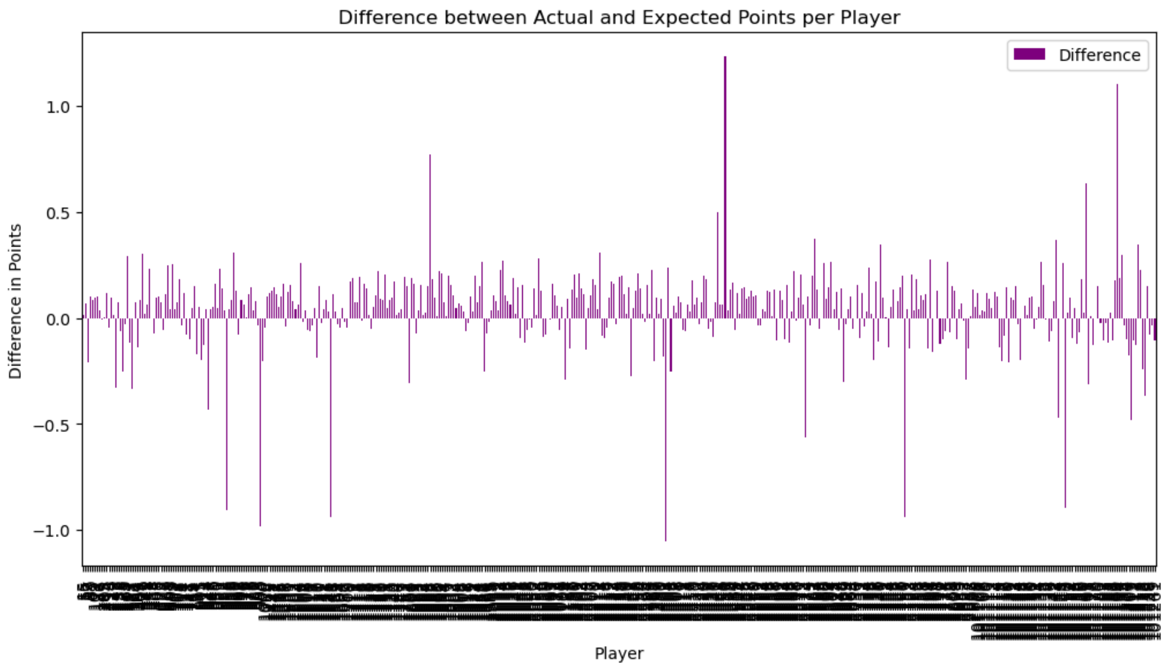
In [42]:

```
# Calculate overall averages
overall_avg_xp = season_1['xP_AVG'].mean()
overall_avg_actual = season_1['POINTS'].mean()
print(f"Overall Average Expected Points: {overall_avg_xp:.2f}")
print(f"Overall Average Actual Points: {overall_avg_actual:.2f}")

# Calculate averages for each player
player_avg_xp = season_1.groupby('PLAYER_ID')['xP_AVG'].mean()
player_avg_actual = season_1.groupby('PLAYER_ID')['POINTS'].mean()

# Create a DataFrame for comparison
comparison_df = pd.DataFrame({
    'Player': player_avg_xp.index,
    'Expected Points': player_avg_xp.values,
    'Actual Points': player_avg_actual.values
})
```

```
# Calculate the difference
comparison_df['Difference'] = comparison_df['Actual Points'] - comparison

print(comparison_df)

# Plot the comparison
comparison_df.plot(kind='bar', x='Player', y=['Expected Points', 'Actual
plt.title('Expected vs Actual Points per Player')
plt.xlabel('Player')
plt.ylabel('Points')
plt.show()

# Plot the differences
comparison_df.plot(kind='bar', x='Player', y='Difference', figsize=(12, 6
plt.title('Difference between Actual and Expected Points per Player')
plt.xlabel('Player')
plt.ylabel('Difference in Points')
plt.show()
```

```
Overall Average Expected Points: 0.89
Overall Average Actual Points: 0.98
     Player  Expected Points  Actual Points  Difference
0        15         0.919883       0.934426    0.014543
1        56         0.891526       0.961609    0.070083
2        57         0.902299       0.692308   -0.209991
3        87         0.949389       1.052632    0.103242
4        89         0.870204       0.953353    0.083148
..      ...              ...            ...         ...
450  101230         0.886611       0.520000   -0.366611
451  101236         0.974462       1.123810    0.149348
452  101238         0.906729       0.830189   -0.076541
453  101249         0.866026       0.833333   -0.032693
454  101261         0.875844       0.770270   -0.105574

[455 rows x 4 columns]
```



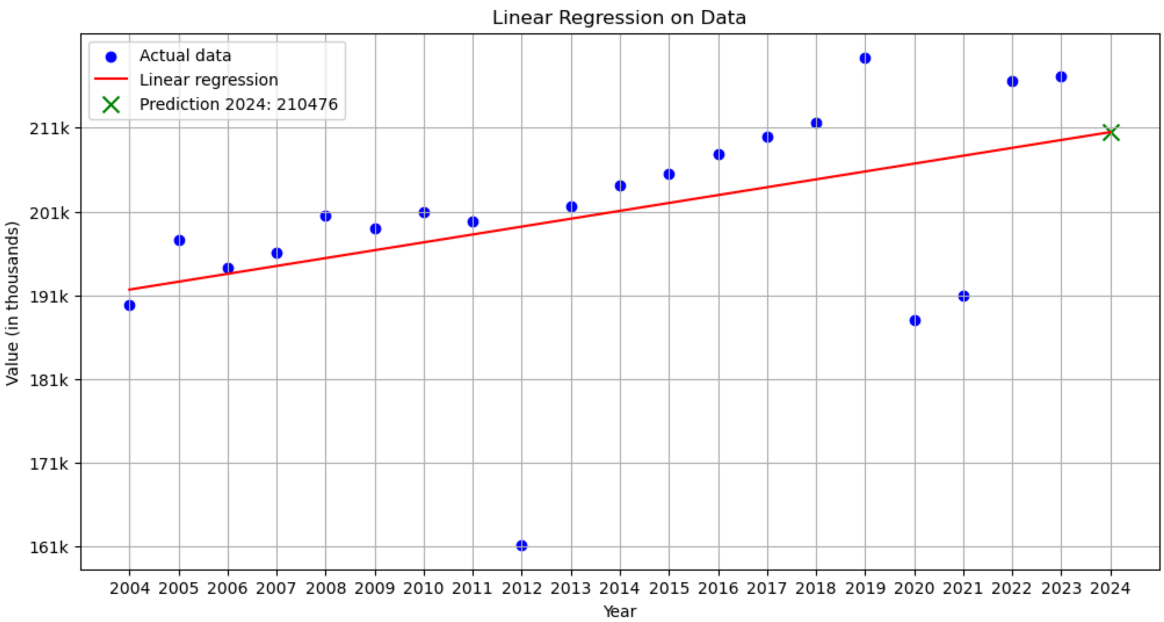



```
In [43]: # Divide the shots for year
shots_4_year = len(data_shots.groupby("SEASON_1"))
shots_4_year

years_dict = {}
for i in range(2004, 2024):
  years_dict[i] = (len(data_shots[data_shots["SEASON_1"] == i]))

print(years_dict)

# check
sum_shots = 0
for year in years_dict.values():
  sum_shots += year
print(f"The sum is {sum_shots}")
```

```
{2004: 189794, 2005: 197609, 2006: 194303, 2007: 196061, 2008: 200469, 200
9: 198992, 2010: 200966, 2011: 199761, 2012: 161205, 2013: 201579, 2014: 2
04126, 2015: 205548, 2016: 207892, 2017: 209928, 2018: 211653, 2019: 21939
8, 2020: 188073, 2021: 190945, 2022: 216644, 2023: 217152}
The sum is 4012098
```

```
In [44]: years = np.array(list(years_dict.keys())).reshape(-1,1)
values = np.array(list(years_dict.values()))

# Create linear regression model
model = LinearRegression()

# Train the model
model.fit(years, values)

# Prediction for 2024
pred_2024 = model.predict([[2024]])

# Regression coefficients
m = model.coef_[0]
q = model.intercept_

# Prediction for all years
```

```
years_extended = np.arange(2004, 2025).reshape(-1, 1)
values_pred = model.predict(years_extended)

# Plot
plt.figure(figsize=(12, 6))
plt.scatter(years, values, color='blue', label='Actual data')
plt.plot(years_extended, values_pred, color='red', label='Linear regressi
plt.scatter([2024], pred_2024, color='green', marker='x', s=100, label=f'
plt.xlabel('Year')
plt.ylabel('Value (in thousands)')
plt.title('Linear Regression on Data')
plt.xticks(np.arange(2004, 2025, 1))
plt.yticks(np.arange(min(values) // 1000 * 1000, max(values) // 1000 * 10
           labels=[f'{int(x/1000)}k' for x in np.arange(min(values) // 10
plt.legend()
plt.grid(True)
plt.show()

print("Prediction for 2024:", round(pred_2024[0],2))
print("Slope (m):", m)
print("Intercept (q):", q)
```




```
Prediction for 2024: 210475.82
Slope (m): 940.087218045113
Intercept (q): -1692260.713533835
```

Heat Maps

In [45]:
```
# creating a list with all the games's id
games_ids = []
games_ids = data_shots["GAME_ID"].unique().tolist()
```

In [46]:
```
# Creating a new draw_curt for drawing the field for the heat map, that i
def draw_court2(ax=None, color='black', lw=2, outer_lines=False, interval
    if ax is None:
        ax = plt.gca()

    # Create the basketball hoop
    hoop = Circle((0, 0), radius=feet_to_m(7.5) / 3, linewidth=lw, color=
```

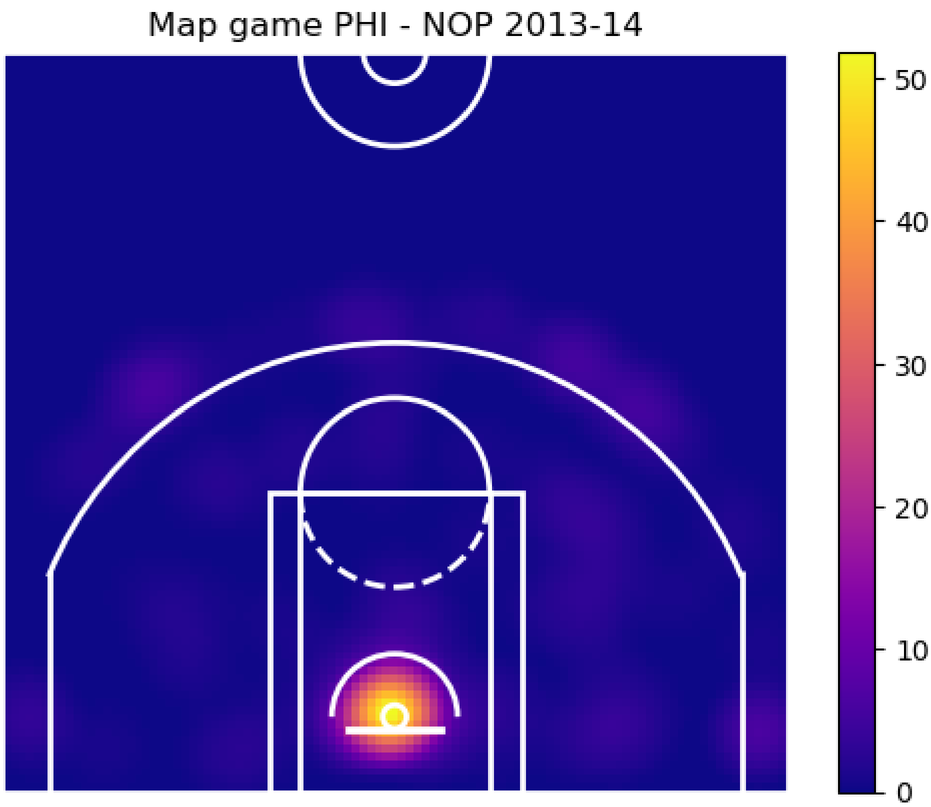
```
    # Create backboard
    backboard = Rectangle((feet_to_m(-30) / 3, -feet_to_m(7.5) / 3), feet

    # The paint
    # Create the outer box 0f the paint, width=16ft, height=19ft
    outer_box = Rectangle((feet_to_m(-80) / 3, -feet_to_m(47.5) / 3), fee
    # Create the inner box of the paint, widt=12ft, height=19ft
    inner_box = Rectangle((feet_to_m(-60) / 3, -feet_to_m(47.5) / 3), fee

    # Create free throw top arc
    top_free_throw = Arc((0, feet_to_m(142.5) / 3), feet_to_m(120) / 3, f
    # Create free throw bottom arc
    bottom_free_throw = Arc((0, feet_to_m(142.5) / 3), feet_to_m(120) / 3
    # Restricted Zone, it is an arc with 4ft radius from center of the ho
    restricted = Arc((0, 0), feet_to_m(80) / 3, feet_to_m(80) / 3, theta1

    # Three point line
    # Create the side 3pt lines, they are 14ft long before they begin to
    corner_three_a = Rectangle((feet_to_m(-220) / 3, -feet_to_m(47.5) / 3
    corner_three_b = Rectangle((feet_to_m(220) / 3, -feet_to_m(47.5) / 3)
    # 3pt arc - center of arc will be the hoop, arc is 23'9" away from ho
    three_arc = Arc((0, 0), feet_to_m(475) / 3, feet_to_m(475) / 3, theta

    # Center Court
    center_outer_arc = Arc((0, feet_to_m(422.5) / 3), feet_to_m(120) / 3,
    center_inner_arc = Arc((0, feet_to_m(422.5) / 3), feet_to_m(40) / 3,

    court_elements = [hoop, backboard, outer_box, inner_box, top_free_thr
                      bottom_free_throw, restricted, corner_three_a,
                      corner_three_b, three_arc, center_outer_arc,
                      center_inner_arc]
    if outer_lines:
        # Draw the half court line, baseline and side out bound lines
        outer_lines = Rectangle((feet_to_m(-250) / 3, -feet_to_m(47.5) /
        court_elements.append(outer_lines)

    for element in court_elements:
        ax.add_patch(element)

    ax.set_aspect('equal', adjustable='box')
    ax.set_xlim(feet_to_m(-250) / 3, feet_to_m(250) / 3)
    ax.set_ylim(-feet_to_m(47.5) / 3, feet_to_m(422.5) / 3)

    return ax
```

In [50]:
```
# Heat map for a random match

from random import choices

game_id =  choices(games_ids)[0]

# Necessary for the intestation of the graph to understand which match we
shots2 = data_shots2[data_shots2['GAME_ID'] == game_id]
home = shots2['HOME_TEAM'].iloc[0]
away = shots2['AWAY_TEAM'].iloc[0]
season = shots2['SEASON_2'].iloc[0]

# Creations of two parallel list with the coordinates of x and y
coordinates = shots2[["LOC_X", "LOC_Y"]]
```

```
x = list(coordinates["LOC_X"].values)
y = list(coordinates["LOC_Y"].values)

#  Each y must be subtracted by 5.2, for fitting properly with the draw o
for i in range(len(y)):
  y[i] -= 5.2

# x,y min and max, necessary for draw properly the heat map along all the
min_x = -24
max_x = 24
min_y = -5
max_y = 45

# Calculating standard deviations and number of points, for calculating t
sigma_x = np.std(x)
sigma_y = np.std(y)
n = len(x)

# Silverman's rule of thumb for bandwidth selection
h = 1.06 * min(sigma_x, sigma_y) * n**(-1/6)
print(f"Calculated bandwidth: {h}")

# Lenght of a rectangle of the Heat map
grid_size = 0.5

# Constructing the Mesh Grid
x_grid = np.arange(min_x - h, max_x + h, grid_size)
y_grid = np.arange(min_y - h, max_y + h, grid_size)
x_mesh, y_mesh = np.meshgrid(x_grid, y_grid)

# Calculating Centre-Points
xc = x_mesh + (grid_size / 2)
yc = y_mesh + (grid_size / 2)


# Define the Quartic Kernel Density Estimator
def kde_quartic(d, h):
  dn = d / h
  S = (15 / 16) * (1 - dn**2) **2
  return S

# Calculate intensity values for each point in the grid
intensity_list = []
for j in range(len(xc)):
  intensity_row = []
  for k in range(len(xc[0])):
    kde_value_list = []
    for i in range(len(x)):
      d = math.sqrt((xc[j][k] - x[i])**2 + (yc[j][k] - y[i])**2)
      if d <= h:
        s = kde_quartic(d, h)
      else:
        s = 0
      kde_value_list.append(s)
    s_total = sum(kde_value_list)
    intensity_row.append(s_total)
  intensity_list.append(intensity_row)

# Converting the list of intensity into an array because it's necessary a
intensity = np.array(intensity_list)
```

```
plt.pcolormesh(x_mesh, y_mesh, intensity, cmap = 'plasma')

# Draw field and title of the graph
draw_court2(plt.gca(), color='white', lw=2, outer_lines=True)
plt.title(f"Map game {home} - {away} {season}")
plt.axis('off')
plt.colorbar()
plt.figure(dpi=100)
plt.show()
```

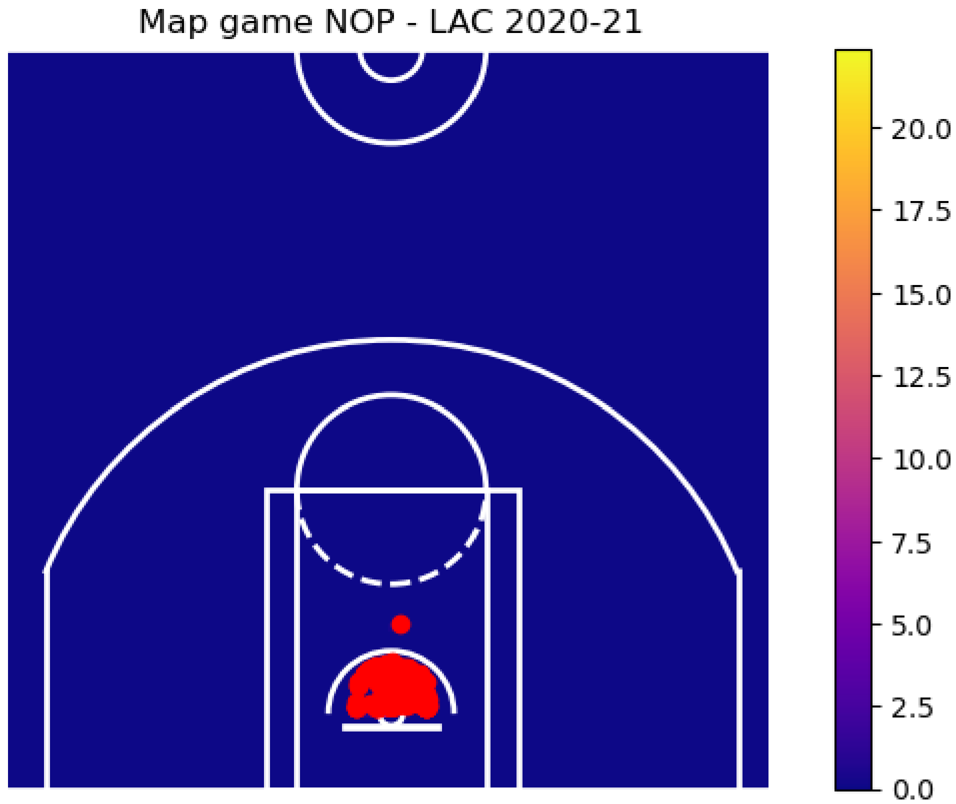
```
Calculated bandwidth: 4.0427828470608755
```




```
<Figure size 640x480 with 0 Axes>
```

In [48]:

```
# This piece of code is for see also the plot of the points on the field
intensity = np.array(intensity_list)
plt.pcolormesh(x_mesh, y_mesh, intensity, cmap = 'plasma')
plt.plot(x, y, 'ro')
draw_court2(plt.gca(), color='white', lw=2, outer_lines=True)
plt.title(f"Map game {home} - {away} {season}")
plt.axis('off')
plt.colorbar()
plt.figure(dpi=100)
plt.show()
```



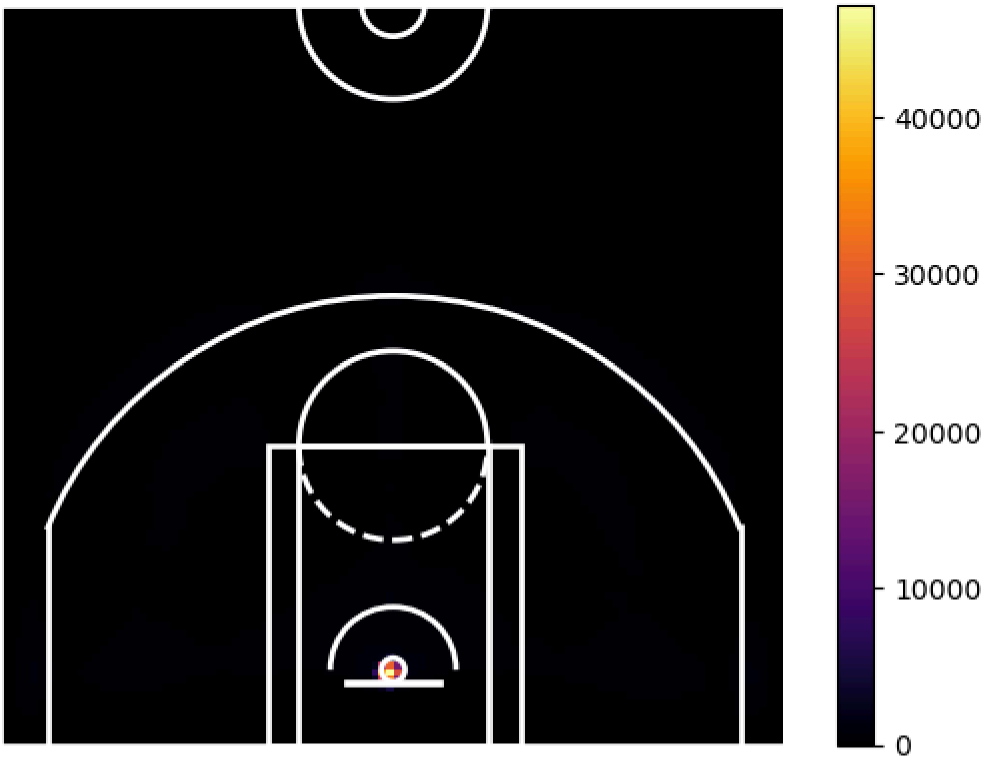
<Figure size 640x480 with 0 Axes>

In [49]:
```
# Heat map of the shots in a season, but with (obvious) bad results

season =  2005

shots2 = data_shots2[data_shots2['SEASON_1'] == season]

sportive_season = shots2['SEASON_2'].iloc[0]

coordinates = shots2[["LOC_X", "LOC_Y"]]
x = list(coordinates["LOC_X"].values)
y = list(coordinates["LOC_Y"].values)

for i in range(len(y)):
  y[i] -= 5.2

min_x = -24
max_x = 24
min_y = -5
max_y = 45

sigma_x = np.std(x)
sigma_y = np.std(y)
n = len(x)

h = 1.06 * min(sigma_x, sigma_y) * n**(-1/6)
print(f"Calculated bandwidth h: {h}")

grid_size = 0.5

x_grid = np.arange(min_x - h, max_x + h, grid_size)
y_grid = np.arange(min_y - h, max_y + h, grid_size)
x_mesh, y_mesh = np.meshgrid(x_grid, y_grid)
```

```python
xc = x_mesh + (grid_size / 2)
yc = y_mesh + (grid_size / 2)

def kde_quartic(d, h):
  dn = d / h
  S = (15 / 16) * (1 - dn**2) **2
  return S

intensity_list = []
for j in range(len(xc)):
  intensity_row = []
  for k in range(len(xc[0])):
    kde_value_list = []
    for i in range(len(x)):
      d = math.sqrt((xc[j][k] - x[i])**2 + (yc[j][k] - y[i])**2)
      if d <= h:
        s = kde_quartic(d, h)
      else:
        s = 0
      kde_value_list.append(s)
    s_total = sum(kde_value_list)
    intensity_row.append(s_total)
  intensity_list.append(intensity_row)

intensity = np.array(intensity_list)
plt.pcolormesh(x_mesh, y_mesh, intensity, cmap = 'inferno')
draw_court2(plt.gca(), color='white', lw=2, outer_lines=True)
plt.title(f"Map game of the sportive season {sportive_season}")
plt.axis('off')
plt.colorbar()
plt.figure(dpi=100)
plt.show()

p_Layups = ((len(shots2[shots2['SHOT_CATEGORY'] == "Layups"]))/len(shots2
p_Dunks = ((len(shots2[shots2['SHOT_CATEGORY'] == "Dunks"]))/len(shots2))
p_Jump = ((len(shots2[shots2['SHOT_CATEGORY'] == "Jump Shots"]))/len(shot
p_Hook = ((len(shots2[shots2['SHOT_CATEGORY'] == "Hook Shots"]))/len(shot
p_Bank = ((len(shots2[shots2['SHOT_CATEGORY'] == "Bank Shots"]))/len(shot
p_Tip = ((len(shots2[shots2['SHOT_CATEGORY'] == "Tip-ins"]))/len(shots2))
p_No = ((len(shots2[shots2['SHOT_CATEGORY'] == "No_Shot"]))/len(shots2))

print(f"The % of the different types of shots for the season: {sportive_s
print(f"Layups: {p_Layups:.2f}%")
print(f"Dunks: {p_Dunks:.2f}%")
print(f"Jump Shots: {p_Jump:.2f}%")
print(f"Hook Shots: {p_Hook:.2f}%")
print(f"Bank Shots: {p_Bank:.2f}%")
print(f"Tip-ins: {p_Tip:.2f}%")
print(f"No Shot: {p_No:.2f}%")

print("\nAs expacted, the most populars are the type of shoot that are cl
```

```
Calculated bandwidth h: 1.1693601999628551
```

Map game of the sportive season 2004-05




```
<Figure size 640x480 with 0 Axes>
The % of the different types of shots for the season: 2004-05 are:
Layups: 23.75%
Dunks: 5.07%
Jump Shots: 66.22%
Hook Shots: 2.92%
Bank Shots: 0.00%
Tip-ins: 2.04%
No Shot: 0.00%

As expacted, the most populars are the type of shoot that are closer to th
e basket, so in the heat map only them are clearly visible
```