In [35]:
```
import numpy as np
import torch
import torchvision.transforms as T
import cv2
import matplotlib.pyplot as plt
from PIL import Image
from torchvision.models.detection import maskrcnn_resnet50_fpn, MaskRCNN_ResNet50_F
from torchvision import models
import torch.hub
import tensorflow as tf

from tensorflow import keras
from tensorflow.keras import layers
```

Volume Estimation

In [36]:
```
# Load MiDaS model for depth estimation
model_type = "DPT_Large"  # Options: "DPT_Large", "DPT_Hybrid", "MiDaS_small"
midas = torch.hub.load("intel-isl/MiDaS", model_type)

device = torch.device("cuda") if torch.cuda.is_available() else torch.device("cpu")
midas.to(device)
midas.eval()

# Load MiDaS transforms
midas_transforms = torch.hub.load("intel-isl/MiDaS", "transforms")
transform_midas = midas_transforms.dpt_transform if model_type in ["DPT_Large", "DP

# Load Mask R-CNN for segmentation
maskrcnn_model = maskrcnn_resnet50_fpn(weights=MaskRCNN_ResNet50_FPN_Weights.DEFAUL
maskrcnn_model.to(device)
maskrcnn_model.eval()

# Function for loading and preprocessing the image
def load_and_preprocess_image(image_path):
    # Load and transform image for depth estimation
    img = cv2.imread(image_path)
    print("Original RGB Pixel Array of Input Image:")
    print(img)
    img_rgb = cv2.cvtColor(img, cv2.COLOR_BGR2RGB)
    print("Image converted to RGB Pixel Array:")
    print(img_rgb)
    input_batch = transform_midas(img_rgb).to(device)
    return img, img_rgb, input_batch



# Function for depth estimation
def estimate_depth(input_batch):
    with torch.no_grad():
        prediction = midas(input_batch)
        depth = torch.nn.functional.interpolate(
            prediction.unsqueeze(1),
```

```
            size=input_batch.shape[2:],
            mode="bicubic",
            align_corners=False,
        ).squeeze()
    depth_map = depth.cpu().numpy()
    print("Depth Map:")
    print(depth_map)
    return depth_map


# Function for segmentation with Mask R-CNN
def segment_objects(image_rgb):
    transform = T.ToTensor()
    img_tensor = transform(image_rgb).to(device)
    with torch.no_grad():
        predictions = maskrcnn_model([img_tensor])
    return predictions[0]

# Function to calculate volume
def calculate_volume(depth_map, masks, pixel_to_cm, depth_scale, threshold=0.5):
    volumes = []
    depth_map_resized = cv2.resize(depth_map, (masks.shape[-1], masks.shape[-2]))
    print("Scaled Depth Map:")
    print(depth_map_resized)

    for i, mask in enumerate(masks):
        mask_binary = (mask[0].cpu().numpy() > threshold).astype(np.uint8)
        mask_resized = cv2.resize(mask_binary, depth_map_resized.shape[::-1])
        print(f"Mask for Object {i+1}:")
        print(mask_resized)

        depth_masked = depth_map_resized * mask_resized
        print(f"Masked Depth Map for Object {i+1}:")
        print(depth_masked)

        area_pixels = np.sum(mask_resized)
        area_cm2 = area_pixels * (pixel_to_cm ** 2)
        print(f"Object {i+1} - Area in Pixels: {area_pixels}")
        print(f"Object {i+1} - Area in cm²: {area_cm2:.2f}")

        depth_values = depth_map_resized[mask_resized > 0]
        height_cm = np.mean(depth_values) * depth_scale if depth_values.size > 0 el
        print(f"Object {i+1} - Height in cm: {height_cm:.2f}")

        volume = area_cm2 * height_cm
        volumes.append(volume)
    return volumes
```

```
Using cache found in C:\Users\User/.cache\torch\hub\intel-isl_MiDaS_master
Using cache found in C:\Users\User/.cache\torch\hub\intel-isl_MiDaS_master
```

In [71]:

```
# Main Execution
# Function to calculate the depth scale from known real-world reference
def calculate_depth_scale(depth_map, known_distance_cm):
    """
    Calculate depth scale from a known real-world distance.
```

```
    :param depth_map: The depth map from MiDaS.
    :param known_distance_cm: The real-world distance in centimeters (reference).
    :return: Depth scale factor.
    """
    # Calculate the average normalized depth value in the depth map
    D_normalized = np.mean(depth_map)

    # Compute the depth scale factor
    depth_scale = known_distance_cm / D_normalized
    return depth_scale

# Example: Known real-world distance for calibration
known_distance_cm = 20  # Replace with the actual real-world distance in cm

# Estimate depth
depth_map = estimate_depth(input_batch)

# Calculate depth scale using the known reference distance
depth_scale = calculate_depth_scale(depth_map, known_distance_cm)
print(f"Calculated Depth Scale: {depth_scale:.2f} cm per normalized unit")
image_path = "C:/123/SRMAP/Semester 7/DIP Lab Project/Images to test on my own/gree
pixel_to_cm = 0.001  # Example scaling factor

img, img_rgb, input_batch = load_and_preprocess_image(image_path)

# Estimate depth
depth_map = estimate_depth(input_batch)

# Segment objects
predictions = segment_objects(img_rgb)
masks = predictions['masks']

# Calculate and print volumes
volumes_cm3 = calculate_volume(depth_map, masks, pixel_to_cm=pixel_to_cm, depth_sca
for i, volume in enumerate(volumes_cm3):
    print(f"Estimated Volume for Object {i+1}: {volume:.2f} cm³")
```

```
Depth Map:
[[ 0.5570541   0.5750065   0.5902854  ...  0.7861476   0.83483773
   0.93971634]
 [ 0.5278547   0.55919033  0.55743945 ...  0.7311828   0.76176673
   0.6857177 ]
 [ 0.5629826   0.55163306  0.5580319  ...  0.7129846   0.70747626
   0.67848146]
 ...
 [15.302324   15.387365   15.409512   ... 17.927486   17.964722
  17.930979  ]
 [15.396764   15.434588   15.4678     ... 17.957523   18.03654
  18.004076  ]
 [15.431556   15.484664   15.476487   ... 18.02566    18.055893
  17.892586  ]]
Calculated Depth Scale: 2.11 cm per normalized unit
Original RGB Pixel Array of Input Image:
[[[255 255 255]
  [255 255 255]
  [255 255 255]
  ...
  [255 255 255]
  [255 255 255]
  [255 255 255]]

 [[255 255 255]
  [255 255 255]
  [255 255 255]
  ...
  [255 255 255]
  [255 255 255]
  [255 255 255]]

 [[255 255 255]
  [255 255 255]
  [255 255 255]
  ...
  [255 255 255]
  [255 255 255]
  [255 255 255]]

 ...

 [[255 255 255]
  [255 255 255]
  [255 255 255]
  ...
  [255 255 255]
  [255 255 255]
  [255 255 255]]

 [[255 255 255]
  [255 255 255]
  [255 255 255]
  ...
  [255 255 255]
  [255 255 255]
```

```
  [255 255 255]]

 [[255 255 255]
  [255 255 255]
  [255 255 255]
  ...
  [255 255 255]
  [255 255 255]
  [255 255 255]]]
Image converted to RGB Pixel Array:
[[[255 255 255]
  [255 255 255]
  [255 255 255]
  ...
  [255 255 255]
  [255 255 255]
  [255 255 255]]

 [[255 255 255]
  [255 255 255]
  [255 255 255]
  ...
  [255 255 255]
  [255 255 255]
  [255 255 255]]

 [[255 255 255]
  [255 255 255]
  [255 255 255]
  ...
  [255 255 255]
  [255 255 255]
  [255 255 255]]

 ...

 [[255 255 255]
  [255 255 255]
  [255 255 255]
  ...
  [255 255 255]
  [255 255 255]
  [255 255 255]]

 [[255 255 255]
  [255 255 255]
  [255 255 255]
  ...
  [255 255 255]
  [255 255 255]
  [255 255 255]]

 [[255 255 255]
  [255 255 255]
  [255 255 255]
  ...
```

```
  [255 255 255]
  [255 255 255]
  [255 255 255]]]
Depth Map:
[[ 0.5570541   0.5750065   0.5902854  ...  0.7861476   0.83483773
   0.93971634]
 [ 0.5278547   0.55919033  0.55743945 ...  0.7311828   0.76176673
   0.6857177 ]
 [ 0.5629826   0.55163306  0.5580319  ...  0.7129846   0.70747626
   0.67848146]
 ...
 [15.302324   15.387365   15.409512   ... 17.927486   17.964722
  17.930979  ]
 [15.396764   15.434588   15.4678     ... 17.957523   18.03654
  18.004076  ]
 [15.431556   15.484664   15.476487   ... 18.02566    18.055893
  17.892586  ]]
Scaled Depth Map:
[[ 0.5570541   0.5570541   0.5570541  ...  0.93971634  0.93971634
   0.93971634]
 [ 0.5570541   0.5570541   0.5570541  ...  0.93971634  0.93971634
   0.93971634]
 [ 0.5570541   0.5570541   0.5570541  ...  0.93971634  0.93971634
   0.93971634]
 ...
 [15.431556   15.431556   15.431556   ... 17.892586   17.892586
  17.892586  ]
 [15.431556   15.431556   15.431556   ... 17.892586   17.892586
  17.892586  ]
 [15.431556   15.431556   15.431556   ... 17.892586   17.892586
  17.892586  ]]
Mask for Object 1:
[[0 0 0 ... 0 0 0]
 [0 0 0 ... 0 0 0]
 [0 0 0 ... 0 0 0]
 ...
 [0 0 0 ... 0 0 0]
 [0 0 0 ... 0 0 0]
 [0 0 0 ... 0 0 0]]
Masked Depth Map for Object 1:
[[0. 0. 0. ... 0. 0. 0.]
 [0. 0. 0. ... 0. 0. 0.]
 [0. 0. 0. ... 0. 0. 0.]
 ...
 [0. 0. 0. ... 0. 0. 0.]
 [0. 0. 0. ... 0. 0. 0.]
 [0. 0. 0. ... 0. 0. 0.]]
Object 1 - Area in Pixels: 2808191
Object 1 - Area in cm²: 2.81
Object 1 - Height in cm: 39.07
Mask for Object 2:
[[0 0 0 ... 0 0 0]
 [0 0 0 ... 0 0 0]
 [0 0 0 ... 0 0 0]
 ...
 [0 0 0 ... 0 0 0]
```

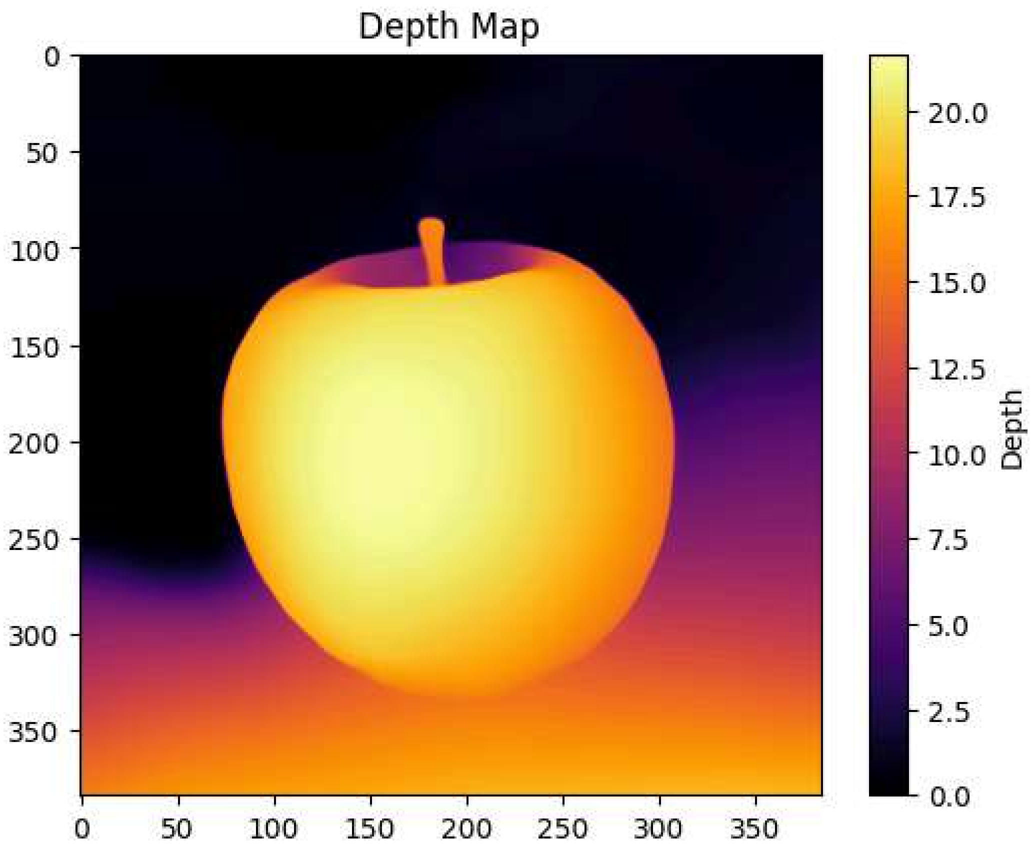
```
 [0 0 0 ... 0 0 0]
 [0 0 0 ... 0 0 0]]
Masked Depth Map for Object 2:
[[0. 0. 0. ... 0. 0. 0.]
 [0. 0. 0. ... 0. 0. 0.]
 [0. 0. 0. ... 0. 0. 0.]
 ...
 [0. 0. 0. ... 0. 0. 0.]
 [0. 0. 0. ... 0. 0. 0.]
 [0. 0. 0. ... 0. 0. 0.]]
Object 2 - Area in Pixels: 2729460
Object 2 - Area in cm²: 2.73
Object 2 - Height in cm: 39.52
Estimated Volume for Object 1: 109.73 cm³
Estimated Volume for Object 2: 107.88 cm³
```

In [72]:

```
# Estimate depth
depth_map = estimate_depth(input_batch)
plt.imshow(depth_map, cmap="inferno")
plt.colorbar(label="Depth")
plt.title("Depth Map")
plt.show()


# Optional: Visualize segmented objects
img_segmented = img_rgb.copy()
for i, box in enumerate(predictions['boxes']):
    if predictions['scores'][i] > 0.5:
        x1, y1, x2, y2 = box.int().cpu().numpy()
        cv2.rectangle(img_segmented, (x1, y1), (x2, y2), (0, 255, 0), 2)
plt.imshow(img_segmented)
plt.title("Segmented Objects")
plt.show()
```

```
Depth Map:
[[ 0.5570541   0.5750065   0.5902854  ...  0.7861476   0.83483773
   0.93971634]
 [ 0.5278547   0.55919033  0.55743945 ...  0.7311828   0.76176673
   0.6857177 ]
 [ 0.5629826   0.55163306  0.5580319  ...  0.7129846   0.70747626
   0.67848146]
 ...
 [15.302324   15.387365   15.409512   ... 17.927486   17.964722
  17.930979  ]
 [15.396764   15.434588   15.4678     ... 17.957523   18.03654
  18.004076  ]
 [15.431556   15.484664   15.476487   ... 18.02566    18.055893
  17.892586  ]]
```



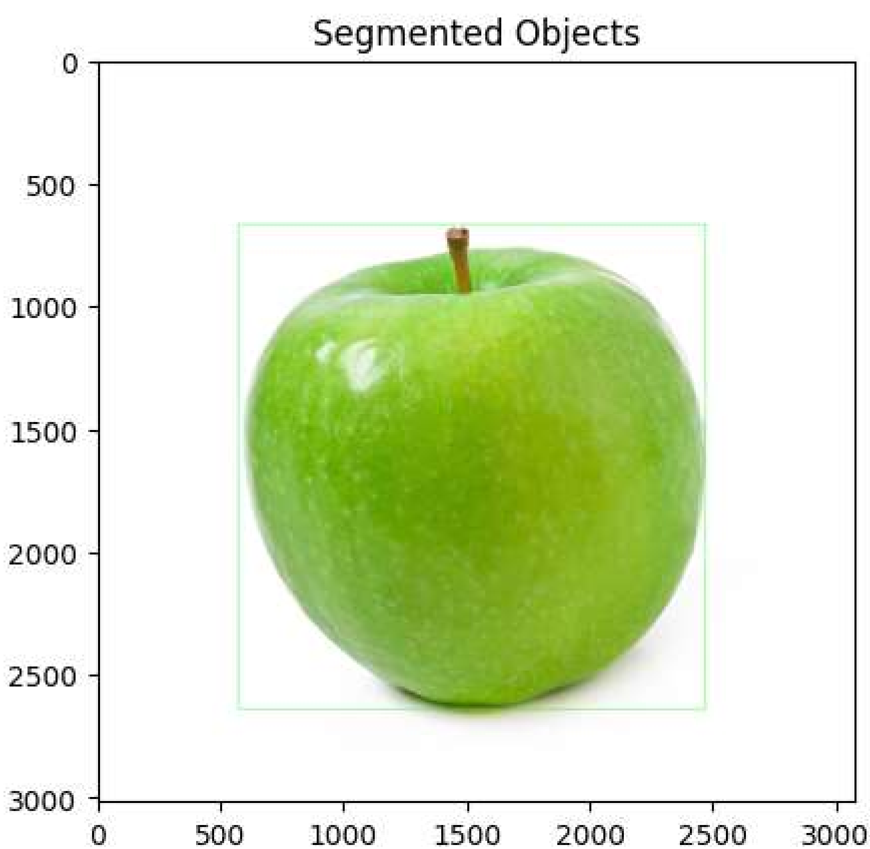


Image Recognition and weight calculation

In [73]:
```
#"C:\123\SRMAP\Semester 7\DIP Lab Project\Image Classification\Fruits_Vegetables"
data_train_path = "C:/123/SRMAP/Semester 7/DIP Lab Project/Image Classification/Fru
data_test_path = "C:/123/SRMAP/Semester 7/DIP Lab Project/Image Classification/Frui
data_val_path = "C:/123/SRMAP/Semester 7/DIP Lab Project/Image Classification/Fruit
```

In [74]:
```
img_width = 180
img_height =180
```

In [75]:
```
data_train = tf.keras.utils.image_dataset_from_directory(
    data_train_path,
    shuffle=True,
    image_size=(img_width, img_height),
    batch_size=32,
    validation_split=False)
```

```
Found 3115 files belonging to 36 classes.
```

In [76]:
```
data_cat = data_train.class_names
```

In [77]:
```
# Load the model
model = tf.keras.models.load_model("C:/123/SRMAP/Semester 7/DIP Lab Project/Image C

# Verify the model structure
model.summary()
```

```
Model: "sequential"
_________________________________________________________________
 Layer (type)                Output Shape              Param #   
=================================================================
 rescaling (Rescaling)       (None, 180, 180, 3)       0         
                                                                 
 conv2d (Conv2D)             (None, 180, 180, 16)      448       
                                                                 
 max_pooling2d (MaxPooling2D  (None, 90, 90, 16)       0         
 )                                                               
                                                                 
 conv2d_1 (Conv2D)           (None, 90, 90, 32)        4640      
                                                                 
 max_pooling2d_1 (MaxPooling  (None, 45, 45, 32)       0         
 2D)                                                             
                                                                 
 conv2d_2 (Conv2D)           (None, 45, 45, 64)        18496     
                                                                 
 max_pooling2d_2 (MaxPooling  (None, 22, 22, 64)       0         
 2D)                                                             
                                                                 
 flatten (Flatten)           (None, 30976)             0         
                                                                 
 dropout (Dropout)           (None, 30976)             0         
                                                                 
 dense (Dense)               (None, 128)               3965056   
                                                                 
 dense_1 (Dense)             (None, 36)                4644      
                                                                 
=================================================================
Total params: 3,993,284
Trainable params: 3,993,284
Non-trainable params: 0
_________________________________________________________________
```

In [80]:

```python
from tensorflow.keras.utils import load_img, img_to_array
```

In [81]:

```python
#"C:\123\SRMAP\Semester 7\DIP Lab Project\Images to test on my own\"
#image = "C:/123/SRMAP/Semester 7/DIP Lab Project/Images to test on my own/apple-25
image = image_path
image = load_img(image, target_size=(img_height,img_width))
img_arr = img_to_array(image)
img_bat = tf.expand_dims(img_arr, 0)
```

In [82]:

```python
predict = model.predict(img_bat)
```

```
WARNING:tensorflow:5 out of the last 5 calls to <function Model.make_predict_functio
n.<locals>.predict_function at 0x0000021994426D30> triggered tf.function retracing.
Tracing is expensive and the excessive number of tracings could be due to (1) creati
ng @tf.function repeatedly in a loop, (2) passing tensors with different shapes, (3)
passing Python objects instead of tensors. For (1), please define your @tf.function
outside of the loop. For (2), @tf.function has reduce_retracing=True option that can
avoid unnecessary retracing. For (3), please refer to https://www.tensorflow.org/gui
de/function#controlling_retracing and https://www.tensorflow.org/api_docs/python/tf/
function for  more details.
1/1 [==============================] - 0s 147ms/step
```

In [83]:
```
score = tf.nn.softmax(predict)
```

In [84]:
```
print('Veg/Fruit in image is {} with accuracy of {:0.2f}'.format(data_cat[np.argmax
```

```
Veg/Fruit in image is apple with accuracy of 35.95
```

Weight Calculation

In [85]:
```
# Define density values for the recognized classes (in g/cm³)
# Note: These values are approximate. Adjust based on more precise references if av
densities = {
    'apple': 0.8,
    'banana': 0.94,
    'beetroot': 0.67,
    'bell pepper': 0.3,
    'cabbage': 0.45,
    'capsicum': 0.3,
    'carrot': 0.65,
    'cauliflower': 0.4,
    'chilli pepper': 0.2,
    'corn': 0.8,
    'cucumber': 0.65,
    'eggplant': 0.4,
    'garlic': 0.59,
    'ginger': 0.75,
    'grapes': 0.95,
    'jalepeno': 0.4,
    'kiwi': 0.88,
    'lemon': 0.92,
    'lettuce': 0.2,
    'mango': 1.0,
    'onion': 0.6,
    'orange': 0.95,
    'paprika': 0.3,
    'pear': 0.59,
    'peas': 0.72,
    'pineapple': 0.98,
    'pomegranate': 1.1,
    'potato': 0.71,
    'raddish': 0.61,
    'soy beans': 0.75,
    'spinach': 0.1,
    'sweetcorn': 0.8,
    'sweetpotato': 0.61,
    'tomato': 0.95,
    'turnip': 0.68,
    'watermelon': 0.95
}
```

In [86]:
```
# Function to calculate weight
def calculate_weight(volume_cm3, predicted_class):
    density = densities.get(predicted_class, None)  # Get density for the class
    if density is None:
        print(f"Density for {predicted_class} not found.")
```

```
        return None
    weight_g = volume_cm3 * density
    return weight_g

# Identify the fruit/vegetable class from the classifier
predicted_class = data_cat[np.argmax(score)]
print(f"Identified Object: {predicted_class}")

# Calculate weight for each detected object
weights = []
for i, volume_cm3 in enumerate(volumes_cm3):
    weight = calculate_weight(volume_cm3, predicted_class)
    if weight is not None:
        print(f"Object {i+1}: {predicted_class}, Volume: {volume_cm3:.2f} cm³, Weig
        weights.append(weight)
    else:
        print(f"Object {i+1}: Volume: {volume_cm3:.2f} cm³. Unable to calculate wei
```

```
Identified Object: apple
Object 1: apple, Volume: 109.73 cm³, Weight: 87.78 g
Object 2: apple, Volume: 107.88 cm³, Weight: 86.30 g
```

In [ ]: