ShoreScan

1. `ImageDatastore`

The `ImageDatastore` class is responsible for managing coastal image data, structured hierarchically by site, camera, year, month, day, time, and image type. It provides functionality to load image metadata, filter low-quality images, and generate summary statistics. The images are identified using metadata extracted from their filenames, such as timestamps, site identifiers, and camera labels. This metadata is organized in a nested dictionary, allowing efficient storage and retrieval of specific subsets of images.

A key feature of the class is its filtering functionality. Methods like `filter_black_images`, `filter_white_images`, and `filter_blurry_images` help remove unsuitable images. For example, `filter_black_images` excludes images that are too dark based on a brightness threshold (default: 50), while `filter_white_images` excludes overly bright images (threshold: 200). The method `filter_blurry_images` uses a Laplacian variance threshold (default values depend on image type, e.g., 20 for `snap`) to detect blurriness. These thresholds are hardcoded but customizable by the user through arguments.

The `ImageDatastore` class also supports copying images into a specified folder, either hierarchically or in a flat structure. Additionally, the `image_stats` method generates useful summaries, such as the total number of images per site or camera, and the distribution of image types. Users can customize some behaviors, such as choosing a root folder for images, but most functionality is predefined for handling typical coastal image datasets. This class ensures that only relevant, high-quality images are passed to subsequent processing stages, enhancing computational efficiency and output reliability.

```
datastore.images[site][camera][year][month][day][time][image_type]
```

`__init__(self, root_folder=None)`
Initializes the datastore with a specified root folder or prompts the user to select one. Sets up a nested dictionary for storing image metadata and a dictionary for camera site information.

`select_folder()`
Opens a dialog box to let the user select a folder. Returns the selected folder path.

`load_images()`
Traverses the directory structure to load image file paths and associated metadata into the datastore based on the filenames.

`copy_images_to_folder(destination_folder=None, hierarchical=False)`
Copies images from the datastore to a specified folder, optionally organizing them into subfolders based on metadata (hierarchical structure).

`list_images(site=None, camera=None, year=None, month=None)`
Lists images filtered by site, camera, year, and/or month. Displays the metadata hierarchy for the specified subset.

`image_stats()`
Displays statistics about the images in the datastore, such as total images, distinct days, and counts by camera and image type.

`get_image_metadata_by_type(image_types, site=None, camera=None)`
Retrieves metadata for specified image types, optionally filtered by site and camera.

`plot_image(image_metadata)`
Displays a single image using its metadata. Includes the camera, type, and date in the title.

`load_camera_sites(csv_file='camera_sites.csv')`
Loads camera site data (latitude, longitude, and angle) from a CSV file and stores it in the `camera_sites` dictionary.

`filter_black_images(threshold=50, image_type=None)`
Removes images that are too dark, based on a brightness threshold.

`filter_white_images(threshold=200, image_type=None)`
Removes images that are too bright, based on a brightness threshold.

`filter_sun_glare(threshold=30, image_type=None, csv_file='camera_sites.csv')`
Detects and removes images affected by sun glare using camera site metadata and solar position calculations using the pvlib library.

`filter_blurry_images(blur_thresholds=None, image_type=None)`
Removes blurry images by checking the variance of the Laplacian, with thresholds specified for different image types.

---

2. `ShorelineDatastore`

The `ShorelineDatastore` class acts as a centralized storage for processed shoreline data. It stores results in a nested dictionary keyed by site, camera, date, and image type. Each entry contains computed shoreline coordinates, bottom boundaries from the segmented SAM mask, watershed segmentation outputs, y-distances, and RMSE values. These metrics are essential for analyzing shoreline dynamics, validating the extraction process, and identifying potential issues.

Users can retrieve stored results for specific images using methods like `get_shoreline_results`. Additionally, the class allows saving shoreline coordinates to text files via `save_shoreline_coords_to_file`. The output filenames are automatically generated based on the metadata of the corresponding image, ensuring easy traceability. This feature enables further analysis or integration with external tools.

Although the `ShorelineDatastore` class does not directly interact with raw image data, it plays a crucial role in organizing and preserving the results of shoreline extraction workflows. Its design ensures consistency and reproducibility, making it a robust component for managing processed data.

`__init__(self)`
Initializes the datastore as a nested dictionary to store shoreline analysis results, organized by site, camera, date, and image type.

`store_shoreline_results(site, camera, year, month, day, time, image_type, shoreline_coords, bottom_boundary, watershed_coords, y_distance, rmse_value)`
Stores computed results for a specific image, including shoreline coordinates, boundaries, watershed results, y-distances, and RMSE values.

`get_shoreline_results(site, camera, year, month, day, time, image_type)`
Retrieves all stored results (e.g., coordinates, boundaries) for a specified image.

`get_shoreline_coords(site, camera, year, month, day, time, image_type)`
Retrieves only the shoreline coordinates for a specified image.

`save_shoreline_coords_to_file(site, camera, year, month, day, time, image_type, output_folder="output")`
Saves shoreline coordinates for a specified image to a text file in the specified folder.

---

3. `ShorelineWorkflow`

The `ShorelineWorkflow` class encapsulates the logic for processing individual images and extracting shoreline features. It uses the SAM (Segment Anything Model) for image segmentation, focusing on surf zones. The workflow is tailored to specific image types, such as `bright` and `timex`, with separate methods implementing customized processing steps. For example, `bright` images undergo boundary extraction, watershed segmentation, and RMSE computation.

Several intermediate plots can be generated during processing, such as overlays of detected boundaries and segmentation masks. This is controlled by the `make_intermediate_plots` argument, which users can toggle when initializing the workflow. Hardcoded thresholds exist for filtering irrelevant regions and computing metrics like y-distance and RMSE. For instance, points where the distance between the SAM shoreline and the determined watershed shoreline exceeds 30 pixels are excluded from the final shoreline.

The workflow outputs its results directly to the `ShorelineDatastore`. Each processing step is designed to handle edge cases, such as missing or invalid data, ensuring robustness. Users primarily interact with the workflow through the main script, where they can specify the image type and enable plotting.

`__init__(self, image_path, image_type, shoreline_datastore, make_intermediate_plots=False)`

Initializes the workflow for processing a single image, linking it to the datastore and specifying whether intermediate plots should be generated.

`_show_plot(self, plt_func, *args, **kwargs)`

Displays a plot if `make_intermediate_plots` is enabled; otherwise, closes the plot.

`process(self)`

Determines the workflow to execute based on the image type (e.g., `bright`, `timex`) and processes the image accordingly.

`_process_bright(self)`

Processes `bright` images. Extracts bottom boundaries, applies watershed segmentation, computes metrics, and stores results in the datastore.

`_process_timex(self)`

Processes `timex` images. Uses previously processed `bright` images for additional boundary information and follows a similar workflow.

`_process_dark(self)`
Placeholder for processing `dark` images. Currently returns a message indicating no workflow is available.

`_process_snap(self)`
Placeholder for processing `snap` images. Currently returns a message indicating no workflow is available.

`_process_var(self)`
Placeholder for processing `var` images. Currently returns a message indicating no workflow is available.

`find_surfzone_coords(image_path, num_points=5, step=200, max_attempts=100, make_plot=False)`
Extracts random points from the largest connected component in the image, representing the surf zone.

`load_and_predict_sam_model(image_path, checkpoint_path, model_type, shoreline_coords, beach_coords=None)`
Loads the SAM model, predicts segmentation masks, and returns the best mask based on scores.

`extract_bottom_boundary_from_mask(mask, make_plot=False, image_path='')`
Extracts the bottom boundary from a segmentation mask using interpolation and smoothing.

`apply_watershed(image_path, bottom_boundary, kernel_size=200, window_size=25, make_plot=False)`
Applies the watershed segmentation algorithm using the bottom boundary as markers and returns the segmented boundary.

`resample_to_boundary(coords_1, coords_2)`
Resamples one set of coordinates to match the length and shape of another using linear interpolation.

`compute_rmse(coords_1, coords_2)`
Computes the RMSE between two sets of coordinates, used to evaluate segmentation accuracy.

`compute_y_distance(coords_1, coords_2)`
Calculates the absolute distance between the y-values of two coordinate sets.

`generate_random_coords_above_line(coords, max_range, min_points, min_y_offset, max_y_offset)`
Generates random coordinates above a given line for use in segmentation workflows.

`plot_image_and_shoreline(image_path, shoreline_coords, watershed_coords, other_coords, y_distance, save_dir)`
Creates a plot showing the image, shoreline boundary, watershed boundary, and optional metrics like y-distance. Saves the plot if a directory is specified.

Main Script

The main script orchestrates the overall process by integrating the three classes. It initializes an `ImageDatastore` instance, loads images, and applies filters to exclude poor-quality data. Users must specify whether to process all images or only new ones by responding to a prompt (`all/new`). If they choose `new`, the script removes already processed images by checking for existing output files (requires both the image and txt file - this can be changed in the future).

After filtering, the script prompts the user to decide whether to enable intermediate plots (`yes/no`). This option affects how the `ShorelineWorkflow` visualizes intermediate steps. The script then processes images of type `bright` and `timex`, passing the filtered datasets and user preferences to the `process_images` function. Users do not need to specify thresholds directly, as these are hardcoded in filtering and workflow methods.

In summary, the main script provides a user-friendly interface for controlling the pipeline. It automates the flow from image loading to data storage, with minimal input required beyond initial choices. Hardcoded thresholds are spread across the filters and workflows, ensuring consistent processing while allowing flexibility for future refinements.

`check_processed_images(image_metadata, pt_dir, output_dir)`
Checks whether a specific image has already been processed by verifying the existence of the associated plot and shoreline point files in the provided directories.

`remove_processed_images(datastore, pt_dir, output_dir)`
Removes already processed images from the ImageDatastore. This prevents redundant processing by filtering out images with corresponding output files already present.

`process_images(datastore, img_type, shoreline_datastore, make_intermediate_plots)`
Processes images of a specified type (bright or timex) using the ShorelineWorkflow. Extracted shoreline data is stored in the ShorelineDatastore. Optionally generates intermediate plots during processing.

`_process_bright(self)`

The `_process_bright` method processes images of the `bright` type and extracts shoreline data through several steps. This method is specifically designed to handle high-contrast images where the surf zone is well-defined. Here's a breakdown of the workflow:

1. **Metadata Extraction**:
    - The method extracts metadata like the month, day, time, year, site, and camera directly from the image file name.
    - These details are later used to store results in the `ShorelineDatastore`.
2. **Surfzone Point Identification**:
    - Locates random points in the surfzone region by:
        - Processing the image to find the largest connected component above an Otsu threshold (typically representing the surfzone).
        - Selects random points from this region to initialize the SAM segmentation model. Minimum 5 points total or every 200 pixels
          These surfzone points provide an anchor for accurate segmentation.
3. **Bottom Boundary Detection**:
    - Determines surfzone points
    - The method attempts to extract the bottom boundary of the surf zone three times using the SAM (Segment Anything Model) for segmentation.
    - Each attempt involves:
        - Using the identified surfzone points to predict a segmentation mask for the surfzone.
        - Extracting the bottom boundary from the mask, which represents the maximum y-values for each x-coordinate in the detected surf zone.
          Only one y-value is allowed for each x-coordinate. This can cause issues when camera angle is along-shore and standing /edge waves are present.
    - This process is repeated three times to reduce noise and improve robustness.
      The median of the three boundaries is calculated as the final bottom boundary.
4. **Watershed Segmentation**:
    - The median bottom boundary from SAM is used as the boundary between ocean and sand and random points above and below are used as input to the watershed segmentation algorithm.
    - This algorithm refines the shoreline boundary by separating it from other regions in the image (e.g., water vs. sand).
5. **Metrics Calculation**:
    - The method calculates the y-distance between the watershed boundary and the SAM-extracted bottom boundary.
    - It also computes the Root Mean Squared Error (RMSE) between these boundaries, providing a quantitative measure of segmentation accuracy.
    - Points where the y-distance exceeds a threshold of 30 pixels are flagged as outliers and excluded from the final shoreline.

6. **Visualization**:
    - If `make_intermediate_plots` is enabled, the method generates visualizations, including:
        - The original image overlaid with the detected boundaries.
        - Intermediate plots for the segmentation mask and watershed results.
    - The final visualization is saved to a directory (`shoreline_plots`).
7. **Data Storage**:
    - The final shoreline coordinates, boundaries, watershed segmentation results, and computed metrics are stored in the `ShorelineDatastore`.

This method is suitable for `bright` images because of their higher contrast and well-defined boundaries, allowing accurate segmentation and analysis.

`_process_timex(self)`

The `_process_timex` method processes images of the `timex` type, which are typically long-exposure images showing time-averaged wave patterns. This method leverages results from previously processed `bright` images to assist in segmentation.

1. **Metadata Extraction**:
    - As with `_process_bright`, metadata like the site, camera, date, and time are extracted from the image file name.
    - These details are used to retrieve previously processed results and store new outputs.
2. **Integration with `bright` Results**:
    - The method retrieves the shoreline coordinates from a corresponding `bright` image stored in the `ShorelineDatastore`.
    - These coordinates provide an initial boundary for further segmentation in the `timex` image.
3. **Bottom Boundary Detection**:
    - Using the coordinates from the `bright` image, random points are generated slightly above the shoreline to mark the surf zone in the `timex` image.
    - SAM segmentation is performed three times to extract bottom boundaries based on these points.
    - Similar to `_process_bright`, the median bottom boundary is computed for stability.
4. **Watershed Segmentation**:
    - The median bottom boundary is refined using the watershed segmentation algorithm, separating the shoreline from other regions.
5. **Metrics Calculation**:

    - The method computes the y-distance and RMSE between the watershed boundary and the SAM-generated boundary.
    - A threshold of 10 pixels for y-distance is applied to filter outliers, which is stricter compared to `_process_bright`.
6. **Visualization**:
    - If `make_intermediate_plots` is enabled, the method generates plots similar to `_process_bright` but also includes the retrieved `bright` shoreline for comparison.
7. **Data Storage**:
    - The shoreline coordinates, segmentation results, and metrics are stored in the `ShorelineDatastore` for further analysis.

This method is particularly effective for `timex` images because they can lack sharp contrasts, making the integration with `bright` results a critical step for accurate shoreline detection.

The shoreline extraction process involves several key steps, each contributing to a robust model capable of accurately identifying and processing shoreline points. Here's an overview of the steps involved, including specific values for parameters used in each stage:

1. **Finding Surfzone Points**: The first step involves extracting up to five random points from the largest connected component within the image, using an Otsu threshold to isolate the surfzone area (the white region). The image is preprocessed by converting it to grayscale and applying a binary threshold. Morphological operations, including opening and erosion with a kernel of size `(25, 100)`, are used to clean the mask. The random points are selected in intervals of 200 pixels along the x-axis. This process is repeated up to 100 times (max attempts) to ensure enough points are found. If successful, the extracted points are used for further segmentation.
2. **SAM Model Prediction**: After identifying the shoreline points, the next step is to use the Segment Anything Model (SAM) for segmentation. The SAM model is loaded using the specified checkpoint, `"segment-anything-main/sam_vit_h_4b8939.pth"`, and set to use the `"vit_h"` model type. The model takes the identified surfzone points, labels them as foreground, and outputs a mask. The best mask is selected based on the highest score, which indicates the model's confidence in the prediction. The model is implemented in PyTorch, and it runs on either a CUDA-enabled GPU or CPU depending on the device availability.
3. **Bottom Boundary Extraction**: Once the mask is generated, the next step is to extract the bottom boundary, which represents the shoreline's lower boundary (maximum y-coordinate for each x-coordinate). This is done by iterating over each x-coordinate and identifying the corresponding y-coordinate where the mask is non-zero. The points are then interpolated to ensure the boundary is continuous and precise.
4. **Watershed Segmentation**: The final segmentation step involves using the watershed algorithm to refine the boundary. The median bottom boundary points are used as the

boundary between sand and water. A dynamic offset from the obtained points based on the exponent of the mean gradient of the window, then smoothed with a Gaussian filter with a kernel size of 200 is used to generate the water markers as above this line and sand markers below. The watershed algorithm is then applied, and the boundary extracted. The result is a set of boundary coordinates that represent an alternative shoreline.

5. **Evaluation**: To evaluate the accuracy of the extracted shorelines, the root mean square error (RMSE) between the watershed coordinates and the median bottom boundary is computed. Additionally, the y-distance between the two sets of coordinates is calculated to assess how closely the watershed boundary aligns with the true shoreline. Points where the y-distance exceeds a threshold (bright: 30, timex: 10 pixels) are flagged as outliers and excluded from the final shoreline obtained from the median bottom boundary of the SAM model.