Perception System Documentation

Overview

The perception system provides VLM-powered semantic scene understanding for robotic manipulation. It uses Google's Gemini Robotics-ER model to dynamically detect objects, infer affordances from visual observation, and compute interaction points for manipulation planning.

Location: src/perception/

Key Features

- **Vision-Language Model Integration**: Uses Gemini Robotics-ER 1.5 or Flash models
- **Dynamic Affordance Prediction**: Infers affordances from appearance without hard-coded rules
- **Async-First Architecture**: Parallel processing for real-time performance
- **3D Spatial Awareness**: Integrates RGB + depth for world-space reasoning
- **Task-Aware Perception**: Prioritizes and refines detections based on task context
- **PDDL Integration**: Tracks object states for automated planning
- **Thread-Safe Registry**: Concurrent access from multiple system components

---

Module Structure

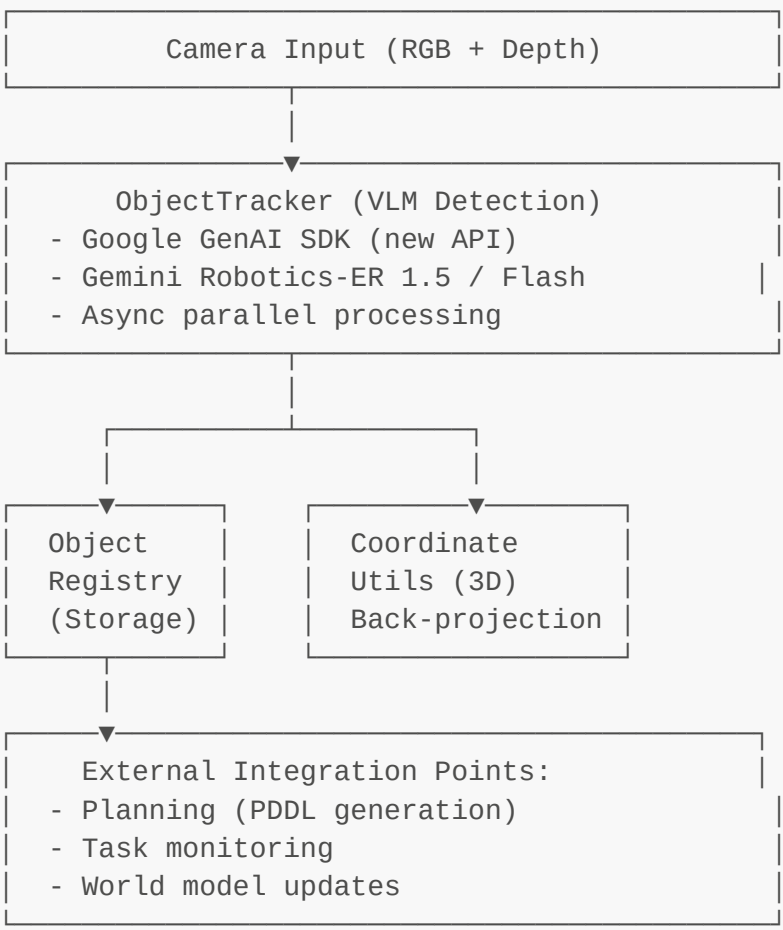
Core Files

- **object_tracker.py**
  - Main VLM-based detection engine using Google GenAI SDK
  - Async streaming and parallel processing
  - Affordance analysis and interaction point detection
  - Registry management and querying
- **object_registry.py**
  - Thread-safe object storage using `threading.RLock`
  - CRUD operations on detected objects
  - Querying by type, affordance, ID
  - JSON serialization/deserialization
- **continuous_tracker.py**
  - Background tracking service
  - Configurable update rate (default 1 Hz)
  - Statistics tracking and callback notifications
- **utils/coordinates.py**
  - 2D/3D coordinate transformations

- Pinhole camera model back-projection
- Batch processing utilities

Architecture

System Flow



Detection Pipeline

Phase 1: Object Name Detection (Streaming)

Initial object detection using streaming API for low latency:

1. Sends RGB image to Gemini VLM
2. Receives object names + bounding boxes in real-time via streaming
3. Format: `OBJECT: red cup | [y1, x1, y2, x2]` (normalized 0-1000 coords)
4. **Timing**: ~2-3 seconds

Phase 2: Parallel Object Analysis (Async)

For each detected object, launches parallel async tasks:

1. **Crops** image to bounding box for efficiency
2. **Resizes** crop to target size (default 512x512)
3. **Analyzes in parallel**:
    - **Affordances**: What actions are possible (graspable, pourable, etc.)
    - **Interaction Points**: Optimal 2D points for each affordance
    - **Properties**: Color, material, size, state
    - **PDDL State Predicates**: Boolean states (clean/dirty, opened/closed)
    - **3D Position**: Back-projects using depth + camera intrinsics

Timing: ~1-2 seconds per object (effective time with parallel processing)

Phase 3: Registry Update

1. Creates `DetectedObject` instances with all analyzed data
2. Adds to thread-safe registry with unique IDs
3. Returns list for immediate use
4. Available for querying by other components

Core Classes

ObjectTracker

Location: src/perception/object_tracker.py

Primary detection engine interfacing with VLM models.

Constructor

```
ObjectTracker(
    api_key: str,
    model_name: str = "auto",           # "auto", "gemini-1.5-flash", etc.
    max_parallel_requests: int = 5,     # Concurrent VLM requests
    crop_target_size: int = 512,        # Resize crops to this size
    enable_affordance_caching: bool = True,  # Cache by object type
    fast_mode: bool = False,            # Skip interaction points
    pddl_predicates: List[str] = None   # PDDL states to track
)
```

Key Methods

```
# Main detection (async)
async def detect_objects(
    color_frame: np.ndarray,
    depth_frame: np.ndarray,
    camera_intrinsics: CameraIntrinsics
) -> List[DetectedObject]
```

```
# Update specific interaction point (async)
async def update_interaction_point(
    object_id: str,
    affordance: str,
    task_context: Optional[str] = None
) -> InteractionPoint

# Registry queries (sync, thread-safe)
def get_object(object_id: str) -> Optional[DetectedObject]
def get_all_objects() -> List[DetectedObject]
def get_objects_by_type(object_type: str) -> List[DetectedObject]
def get_objects_with_affordance(affordance: str) -> List[DetectedObject]

# Configuration
def set_pddl_predicates(predicates: List[str])
```

Usage Example

```
from src.perception import ObjectTracker
from src.camera import RealSenseCamera

# Initialize
tracker = ObjectTracker(
    api_key=api_key,
    fast_mode=False,
    pddl_predicates=["clean", "opened", "filled"]
)
camera = RealSenseCamera(enable_depth=True, auto_start=True)

# Detect objects
color_image, depth_image = camera.get_aligned_frames()
intrinsics = camera.get_camera_intrinsics()
objects = await tracker.detect_objects(color_image, depth_image,
intrinsics)

# Query results
cups = tracker.get_objects_by_type("cup")
graspable = tracker.get_objects_with_affordance("graspable")
```

DetectedObject

Location: src/perception/object_registry.py

Data container representing a detected object.

Attributes

```
class DetectedObject:
    object_type: str                    # Category (e.g., "cup")
    object_id: str                      # Unique ID (e.g., "red_cup_1")
    affordances: set[str]               # {"graspable", "pourable"}
    interaction_points: Dict[str, InteractionPoint]  # Per-affordance
    position_2d: List[int]              # [y, x] normalized 0-1000
    position_3d: np.ndarray             # [x, y, z] in meters
    bounding_box_2d: List[int]          # [y1, x1, y2, x2] normalized
    properties: Dict[str, any]          # Color, material, size, state
    pddl_state: Dict[str, bool]         # {"clean": True, "opened": False}
    confidence: float                   # Detection confidence
    timestamp: float                    # Unix timestamp
```

InteractionPoint

Location: src/perception/object_registry.py

Affordance-specific interaction point for manipulation.

Attributes

```
class InteractionPoint:
    position_2d: List[int]              # [y, x] normalized 0-1000
    position_3d: Optional[np.ndarray]   # [x, y, z] meters
    confidence: float                   # Point confidence (0-1)
    reasoning: str                      # VLM explanation
    alternative_points: List[Dict]      # Backup points
```

DetectedObjectRegistry

Location: src/perception/object_registry.py

Thread-safe storage and query system.

Key Methods

```
# Storage operations (all thread-safe)
def add_object(obj: DetectedObject)
def get_object(object_id: str) -> Optional[DetectedObject]
def update_object(object_id: str, obj: DetectedObject) -> bool
def remove_object(object_id: str) -> bool
def clear()

# Queries
def get_all_objects() -> List[DetectedObject]
def get_objects_by_type(object_type: str) -> List[DetectedObject]
def get_objects_with_affordance(affordance: str) -> List[DetectedObject]
```

```
# Persistence
def save_to_json(output_path: str) -> str
def load_from_json(input_path: str) -> List[DetectedObject]
```

ContinuousObjectTracker

Location: src/perception/continuous_tracker.py

Background service for continuous tracking.

Features

- Async background loop
- Configurable update interval (default 1 Hz)
- Statistics tracking (detection count, average time)
- Callback notifications on updates
- Thread-safe access from multiple consumers

Usage Example

```
from src.perception import ContinuousObjectTracker

# Initialize
tracker = ContinuousObjectTracker(
    api_key=api_key,
    fast_mode=True,
    update_interval=0.5  # 2 Hz
)

# Set frame source
tracker.set_frame_provider(lambda: camera.get_aligned_frames())

# Start background loop
tracker.start()

# Query from other threads
objects = tracker.get_objects_with_affordance("graspable")

# Stop when done
await tracker.stop()
```

Coordinate Systems

1. Normalized Coordinates (0-1000)

- **Used by**: VLM output
- **Format**: `[y, x]` in 0-1000 range

- **Purpose**: Resolution-independent
- **Example**: `[500, 300]` = middle-top-left

2. Pixel Coordinates

- **Used by**: Image processing, depth lookup
- **Format**: `(pixel_y, pixel_x)` in image dimensions
- **Purpose**: Direct array indexing
- **Example**: `(480, 640)` for center of 960x1280 image

3. 3D World Coordinates

- **Used by**: Spatial reasoning, motion planning
- **Format**: `[x, y, z]` in meters (camera frame)
- **Purpose**: Physical world positions
- **Origin**: Camera optical center

Conversion Utilities

Location: src/perception/utils/coordinates.py

```
# 2D conversions
normalized_to_pixel(normalized_pos, image_shape) -> (pixel_y, pixel_x)
pixel_to_normalized(pixel_pos, image_shape) -> [norm_y, norm_x]

# 3D back-projection (requires depth)
compute_3d_position(position_2d, depth_frame, camera_intrinsics) -> [x, y,
z]
project_3d_to_2d(position_3d, camera_intrinsics, image_shape) -> [y, x]

# Batch operations
batch_compute_3d_positions(positions_2d, depth_frame, intrinsics) -> List
get_depth_at_normalized_position(position_2d, depth_frame) -> float
```

Pinhole Camera Model

```
x = (pixel_x - cx) * depth / fx
y = (pixel_y - cy) * depth / fy
z = depth
```

Where `(cx, cy)` = principal point, `(fx, fy)` = focal lengths

Affordance System

Dynamic, Vision-Based Inference

Key Innovation: No hard-coded object-to-affordance mappings

- VLM infers affordances from object appearance
- Works with novel objects not seen during training
- Context-aware (considers task when predicting)

Common Affordances

- **graspable**: Object can be grasped (handles, edges)
- **pourable**: Liquid can be poured from/into object
- **containable**: Object can contain other objects
- **supportable**: Provides flat surface for placing items
- **openable**: Has opening mechanism (doors, lids)
- **pushable**: Can be pushed
- **pullable**: Can be pulled
- **rotatable**: Can be rotated
- **static**: Fixed in environment

Affordance Caching

For efficiency, caches affordances by object type:

```
# First detection of "cup"
cup1 = await tracker.detect_objects(...)  # Full VLM analysis

# Second detection of different cup
cup2 = await tracker.detect_objects(...)  # Uses cached affordances
```

Benefits:

- Reduced API calls and cost
- Lower latency for repeated types
- Per-instance interaction points still computed

Configuration:

```
tracker = ObjectTracker(enable_affordance_caching=True)  # Default
```

PDDL Integration

State Tracking

Configure PDDL predicates for planning:

```
# Set predicates to track
tracker.set_pddl_predicates(["clean", "opened", "filled", "heavy"])
```

```
# Detect objects
objects = await tracker.detect_objects(color, depth, intrinsics)

# Objects include PDDL state
for obj in objects:
    print(obj.pddl_state)
    # {"clean": True, "opened": False, "filled": True, "heavy": False}
```

Planning System Integration

Flow:

1. **Task Analysis** → Identifies required predicates
2. **Perception** → Detects objects with PDDL states
3. **Domain Maintenance** → Updates PDDL domain from observations
4. **PDDL Generation** → Uses object instances and states

See planning.md for full integration details.

Performance

Timing Benchmarks

Typical Detection (10 objects):

- Object name detection: ~2-3 seconds (streaming)
- Parallel object analysis: ~8-15 seconds (async)
- **Total**: ~10-20 seconds
- **Per-object effective**: ~1-2 seconds (with parallelism)

Fast Mode (skip interaction points):

- **Total**: ~5-10 seconds for 10 objects
- **Speedup**: 2-3x faster

Optimization Strategies

1. **Image Encoding Caching**: Reuses encoded images for parallel requests
2. **Affordance Caching**: Caches by object type
3. **Bounding Box Cropping**: Reduces VLM input size
4. **Async/Await**: True parallel processing
5. **Fast Mode**: Skips interaction points when not needed
6. **Crop Resizing**: Configurable target size (smaller = faster)

Configuration

Perception Config (config/perception_config.yaml):

```
detection_rate: 5  # Hz
max_objects: 30
confidence_threshold: 0.6
near_distance_threshold: 0.3  # meters
async_processing: true
num_workers: 2
```

Usage Examples

Basic Detection

```
from src.perception import ObjectTracker
from src.camera import RealSenseCamera

# Setup
tracker = ObjectTracker(api_key=api_key)
camera = RealSenseCamera(enable_depth=True, auto_start=True)

# Detect
color, depth = camera.get_aligned_frames()
intrinsics = camera.get_camera_intrinsics()
objects = await tracker.detect_objects(color, depth, intrinsics)

# Query
for obj in objects:
    print(f"{obj.object_id}: {obj.affordances}")
    print(f"  Position: {obj.position_3d}")
    print(f"  Interaction points: {list(obj.interaction_points.keys())}")
```

Continuous Tracking

```
from src.perception import ContinuousObjectTracker

# Setup
tracker = ContinuousObjectTracker(
    api_key=api_key,
    update_interval=1.0,  # 1 Hz
    fast_mode=False
)

# Frame source
def get_frames():
    color, depth = camera.get_aligned_frames()
    intrinsics = camera.get_camera_intrinsics()
    return color, depth, intrinsics

tracker.set_frame_provider(get_frames)
```

```
tracker.start()

# Use from other threads
while task_running:
    cups = tracker.get_objects_by_type("cup")
    if cups:
        nearest = min(cups, key=lambda o: np.linalg.norm(o.position_3d))
        print(f"Nearest cup: {nearest.object_id}")
    await asyncio.sleep(0.5)

await tracker.stop()
```

Task-Aware Perception

```
# Set task context
task = "Pick up the red cup and pour water into the bowl"

# Detection considers task
objects = await tracker.detect_objects(color, depth, intrinsics)

# Update interaction point for task
cup = tracker.get_objects_by_type("cup")[0]
grasp_point = await tracker.update_interaction_point(
    cup.object_id,
    "graspable",
    task_context=task
)
print(f"Task-specific grasp: {grasp_point.position_3d}")
print(f"Reasoning: {grasp_point.reasoning}")
```

Registry Persistence

```
# Save detected objects
json_path = tracker.registry.save_to_json("detected_objects.json")
print(f"Saved to {json_path}")

# Load later
loaded = tracker.registry.load_from_json("detected_objects.json")
print(f"Loaded {len(loaded)} objects")
```

Troubleshooting

No Objects Detected

Causes: Poor lighting, objects too small, low confidence threshold

Solutions:

```
# Lower confidence threshold
tracker = ObjectTracker(confidence_threshold=0.4)  # Default 0.6

# Improve lighting, move camera closer, use higher resolution
```

Incorrect 3D Positions

Causes: Misaligned frames, invalid depth, incorrect intrinsics

Solutions:

```
# Ensure alignment
camera = RealSenseCamera(enable_depth=True)  # Alignment automatic

# Verify intrinsics
intrinsics = camera.get_camera_intrinsics()
print(f"fx={intrinsics.fx}, fy={intrinsics.fy}")

# Filter invalid depth
depth_image[depth_image == 0] = np.nan
```

Slow Performance

Causes: Too many parallel requests, large images, network latency

Solutions:

```
# Reduce parallel requests
tracker = ObjectTracker(max_parallel_requests=3)  # Default 5

# Smaller crop size
tracker = ObjectTracker(crop_target_size=256)  # Default 512

# Enable fast mode
tracker = ObjectTracker(fast_mode=True)

# Lower resolution
camera = RealSenseCamera(width=640, height=480)
```

API Rate Limiting

Causes: Too many requests, quota exceeded

Solutions:

```

```

```
# Reduce detection rate
tracker = ContinuousObjectTracker(update_interval=2.0)  # 0.5 Hz

# Enable caching
tracker = ObjectTracker(enable_affordance_caching=True)

# Check API key
import os
api_key = os.getenv("GOOGLE_API_KEY")
```

Future Enhancements

Planned Features

1. **Multi-Frame Tracking**: Track identity across frames, estimate velocities
2. **Semantic Relationships**: Detect spatial relations (on, in, near), build scene graphs
3. **Active Perception**: Suggest viewpoints, handle ambiguity, verify hypotheses
4. **Temporal Affordances**: Predict state transitions over time
5. **Hierarchical Models**: Part-based decomposition, articulated objects

---

Related Documentation

- Camera System
- Planning System
- Perception Module README

Last Updated: November 20, 2025