

Heuristics

PhantomLimb

Spatial (joints above floor)

- **Code:**

```
for j in range(24): # Assuming 24 joints: Skeleton0 to Skeleton23
    joint_y = float(row[f'Skeleton{j}_Y'])
    floor_y = float(row['Floor_Y']) if 'Floor_Y' in row else 0.0 # default to 0 if not present

    if joint_y < floor_y:
        joints_below_floor += 1

# Apply heuristic: 0 or 1 joints under floor = good (1), else bad (0)
heuristic_flags.append(1 if joints_below_floor <= 1 else 0)
```

- **Formula:**

$$\text{Heuristic} = \begin{cases} 1, & \text{if } |\{j \in Joints \mid y_j \geq y_{\text{floor}}\}| \geq \text{len}(Joints) - 1 \\ 0, & \text{otherwise} \end{cases}$$

- **Explanation:** We allow a small deviation to account for the reading errors of the Kinect camera. So, we are checking if the skeletal joints are above the floor, allowing one to be misplaced. This accounts for possible errors that were allowed due to manual labeling and Kinect camera misreading.

Temporal (user can reach the bubble generator)

- **Code:**

```
if np.linalg.norm(hip_pos - bubble_generators_pos) <= total_reach + bubble_radius + tolerance:
    reachable_bubbles += 1

# Frame label: 1 if misses 0 or 1 bubbles; 0 if misses 2 or more
frame_labels.append(1 if reachable_bubbles >= 5 else 0)
```

- **Formula:**

$$\text{Heuristic}(s) = \begin{cases} 1, & \left| \{b \in \text{Bubble_gen.} : \text{distance}(s, b) \leq \text{total_reach}(s) + \tau\} \right| \geq \#\text{Bubble_gen.} - 1 \\ 0, & \text{otherwise} \end{cases}$$

- **Explanation:** We calculate the total reach of the user's skeleton (based on the bone lengths derived from the joint positions) and compare it to the total distance from the user's skeleton to the bubble generators.
- **Reason:** We allow a small tolerance (τ) when comparing the total reach with the total distance to account for the Kinect camera misreadings. For the same reason, we also allow the user to miss one bubble generator, as long as the other are in reach.
- **Source:** Data derived tolerance.

PianoTiles

Spatial (feet above floor)

- **Code:**

```
for j in range(24): # Assuming 24 joints: Skeleton0 to Skeleton23
    joint_y = float(row[f'Skeleton{j}_Y'])
    floor_y = float(row['Floor_Y']) if 'Floor_Y' in row else 0.0 # default to 0 if not present

    if joint_y < floor_y:
        joints_below_floor += 1

# Apply heuristic: 0 or 1 joints under floor = good (1), else bad (0)
heuristic_flags.append(1 if joints_below_floor <= 1 else 0)
```

- **Formula:**

$$\text{Heuristic} = \begin{cases} 1, & \text{if } |\{j \in Joints \mid y_j \geq y_{\text{floor}}\}| \geq \text{len}(Joints) - 1 \\ 0, & \text{otherwise} \end{cases}$$

- **Explanation:** We allow a small deviation to account for the reading errors of the Kinect camera. So, we are checking if the skeletal joints are above the floor, allowing one to be misplaced. This accounts for possible errors that were allowed due to manual labeling and Kinect camera misreading.

Temporal (user can reach the dynamic tiles)

- **Code:**

```
if np.linalg.norm(hip_pos - sphere_pos) <= total_reach + bubble_radius:
    reachable_bubbles += 1

# Frame label: 1 if misses 0 or 1 bubbles; 0 if misses 2 or more
frame_labels.append(1 if reachable_bubbles >= 11 else 0)
```

- **Formula:**

$$\text{Heuristic}(s) = \begin{cases} 1, & \left| \{\forall b \in \text{Tiles} : \text{distance}(s, b) \leq \text{total_reach}(s) + \tau\} \right| \geq \#\text{Tiles} - 1 \\ 0, & \text{otherwise} \end{cases}$$

- **Explanation:** We calculate the total reach of the user's skeleton (based on the bone lengths derived from the joint positions) and compare it to the total distance from the user's skeleton to the tiles.
- **Reason:** We allow a small tolerance (Tau) when comparing the total reach with the total distance to account for the Kinect camera misreadings. For the same reason, we also allow the user to miss one tile, as long as the other are in reach.
- **Source:** Data derived tolerance.

Archery

Spatial (left and right feet are facing the same direction)

- **Code:**

```

# Angle between vectors
angle = compute_angle(vec_left, vec_right)

# Determine direction
cross = np.cross(vec_left, vec_right)
direction = np.sign(cross[1]) # positive = outward, negative = inward

# Apply heuristic
if direction >= 0:
    is_valid = angle <= 70
else:
    is_valid = angle <= 50

```

- **Formula:**

$$\text{Heuristic}_{\text{feet}} = \begin{cases} 1, & \text{if } \text{direction}(\text{foot}_L, \text{foot}_R) = \text{inward} \wedge \text{angle}(\text{foot}_L, \text{foot}_R) \leq 50^\circ \\ 1, & \text{if } \text{direction}(\text{foot}_L, \text{foot}_R) = \text{outward} \wedge \text{angle}(\text{foot}_L, \text{foot}_R) \leq 70^\circ \\ 0, & \text{otherwise} \end{cases}$$

- **Constants:** inner/outer angles = **50, 70**
- **Explanation:** We use the vector derived from the feet position to extract the direction towards which the feet are pointing. For the right skeletal configuration, we check that both feet point in the same direction.
- **Source:** Angle was extracted from the data and experiments
- **Reason:** We use the angles to check if the both feet are within normal configuration.

Temporal (aim & face directions are facing the targets)

- **Code:**

```

# At least one target in front (+1)
targets_forward_ok = any(s == 1 for s in nonzero)
# set tdir = 1 if any forward, else -1
tdir = 1 if targets_forward_ok else -1

# Face should match targets; Arrow should match face (or targets)
face_matches_targets = (face_z != 0 and face_z == tdir)
arrow_matches_face = (arrow_z != 0 and arrow_z == face_z)

pred = int(face_matches_targets and arrow_matches_face and targets_forward_ok)

```

- **Formula:**

$$\text{Heuristic} = \begin{cases} 1, & (\text{Face_direction} = \text{Trajectory_direction} = \text{forward}) \wedge (\forall t \in \text{Target} : t \in \text{forward}) \\ 0, & \text{otherwise} \end{cases}$$

- **Explanation:** The Face and Trajectory directions are extracted from a face and trajectory vector respectively, constructed from the collected positions
- **Reason:** We use the directional vectors to check if the face and aiming trajectory are pointing towards the same direction (the user aims in front of his field of view), and to check if the shooting targets are in the direction of the aim trajectory (the users is trying to shoot at the targets not away from them)

Puzzle

Spatial (user within playable area)

- **Code:**

```
# --- Step 2: Check for any missing (empty string) joints ---
has_missing = df[x_cols + y_cols].applymap(lambda val: val == "").any(axis=1)

# ...

# --- Step 4: Check if all joints are within the FOV ---
within_x = df[x_cols].apply(lambda col: col.between(*x_range)).all(axis=1)
within_y = df[y_cols].apply(lambda col: col.between(*y_range)).all(axis=1)
in_fov = within_x & within_y
```

- **Formula:**

$$\text{Heuristic} = \begin{cases} 0, & \neg\exists j \in \text{Skeletal_joints} \\ 1, & \forall j \in \text{Skeletal_joints} : (X_{\min} \leq X_j \leq X_{\max}) \wedge (Y_{\min} \leq Y_j \leq Y_{\max}) \\ 0, & \text{otherwise} \end{cases}$$

- **Explanation:** We use set limits for the X and Y axis to check if the user is within screen bounds while playing.
- **Source:** Data derived limits.

Temporal (user within playable area)

- **Code:**

```
# Missing joints (NaNs in any of the X/Y columns)
has_missing = window[x_cols + y_cols].isna().any(axis=1).any()

# FOV checks
within_x = window[x_cols].apply(lambda col: col.between(*x_range)).all(axis=1).all()
within_y = window[y_cols].apply(lambda col: col.between(*y_range)).all(axis=1).all()

is_valid = not has_missing and within_x and within_y
```

- **Formula:**

$$\text{Heuristic} = \begin{cases} 0, & \neg\exists j \in \text{Skeletal_joints} \\ 1, & \forall j \in \text{Skeletal_joints} : (X_{\min} \leq X_j \leq X_{\max}) \wedge (Y_{\min} \leq Y_j \leq Y_{\max}) \\ 0, & \text{otherwise} \end{cases}$$

- **Explanation:** We use set limits for the X and Y axis to check if the user is within screen bounds while playing.
- **Source:** Data derived limits.

Sea

Spatial (feet joints above floating raft)

- **Code:**

```
# --- Step 1: Relevant foot Y-joints ---
y_cols = ["FootL_Y", "FootR_Y"]

# --- Step 2: Check for missing data (empty strings)
has_missing = df[y_cols].applymap(lambda val: val == "").any(axis=1)

# --- Step 3: Convert to numeric ("" → NaN)
for col in y_cols:
    df[col] = pd.to_numeric(df[col], errors="coerce")

# --- Step 4: Check if both feet are above threshold
above_threshold = df[y_cols].apply(lambda col: col >= foot_threshold).all(axis=1)

# --- Step 5: Heuristic rule ---
df["rule_spatial_feet_sea"] = np.where(has_missing, 0, np.where(above_threshold, 1, 0))
```

- **Formula:**

$$\text{Heuristic} = \begin{cases} 0, & \neg\exists j \in \text{Foot_joints} \\ 1, & \forall j \in \text{Foot_joints} : Y_j \geq Y_{raft} + \tau \\ 0, & \text{otherwise} \end{cases}$$

- **Reason:** To account for Kinect camera misreadings we allowed a set tolerance.
- **Source:** The tolerance (Tau) value is data driven.

Temporal (correct skeletal configuration and placement)

- **Code:**

```
has_missing = footL.isna().any() or footR.isna().any()

if has_missing:
    label = 0
else:
    all_above = (footL >= foot_threshold) & (footR >= foot_threshold)
    label = int(all_above.all())
```

- **Formula:**

$$\text{Heuristic} = \begin{cases} 0, & \neg\exists j \in \text{Foot_joints} \\ 1, & \forall j \in \text{Foot_joints} : Y_j \geq Y_{raft} + \tau \\ 0, & \text{otherwise} \end{cases}$$

- **Reason:** To account for Kinect camera misreadings we allowed a set tolerance.
- **Source:** The tolerance (Tau) value is data driven.

War

Spatial (user within playable area)

- **Code:**

```
# --- Step 2: Check for missing joints ---
has_missing = df[x_cols + y_cols].applymap(lambda val: val == "").any(axis=1)

# ...

# --- Step 4: Check if joints are within the field of view ---
within_x = df[x_cols].apply(lambda col: col.between(*x_range)).all(axis=1)
within_y = df[y_cols].apply(lambda col: col.between(*y_range)).all(axis=1)
in_fov = within_x & within_y
```

- **Formula:**

$$\text{Heuristic} = \begin{cases} 0, & \neg\exists j \in \text{Skeletal_joints} \\ 1, & \forall j \in \text{Skeletal_joints} : (X_{\min} \leq X_j \leq X_{\max}) \wedge (Y_{\min} \leq Y_j \leq Y_{\max}) \\ 0, & \text{otherwise} \end{cases}$$

- **Explanation:** We use set limits for the X and Y axis to check if the user is within screen bounds while playing.
- **Source:** Data derived limits.

Temporal (user within playable area)

- **Code:**

```
--- Step 2: Check for missing joints ---
has_missing = df[x_cols + y_cols].applymap(lambda val: val == "").any(axis=1)

# ...

# --- Step 4: Check if joints are within the field of view ---
within_x = df[x_cols].apply(lambda col: col.between(*x_range)).all(axis=1)
within_y = df[y_cols].apply(lambda col: col.between(*y_range)).all(axis=1)
in_fov = within_x & within_y
```

- **Formula:**

$$\text{Heuristic} = \begin{cases} 0, & \neg\exists j \in \text{Skeletal_joints} \\ 1, & \forall j \in \text{Skeletal_joints} : (X_{\min} \leq X_j \leq X_{\max}) \wedge (Y_{\min} \leq Y_j \leq Y_{\max}) \\ 0, & \text{otherwise} \end{cases}$$

- **Explanation:** We use set limits for the X and Y axis to check if the user is within screen bounds while playing.
- **Source:** Data derived limits.