Object Representation in Angry Birds Game

Shu Lin, Qinjian Zhang, Haifeng Zhang
Peking University
China
{fzlinshu, zqj, pkuzhf}@pku.edu.cn

Abstract

In the Angry Birds game, there are dynamic objects and static objects. Dynamic objects are usually convex polygons, while static objects can be concave. To represent dynamic objects, it is more accurate to use bounding convex polygons (BCPs) than Axis-aligned minimum bounding boxes (AABBs). BCPs can be roughly detected from AABBs, and some of them can be fitted into rectangles. We take another approach to detect concave objects, such as mountains. We apply edge detection and Hough Transform to build polygons to represent mountains. In this paper, we mainly introduce our work on detecting BCPs and representing concave objects.

1 Introduction

Angry Birds AI Challenge is an AI competition, aiming to build AI agents that can play the Angry Birds game. In this game, birds are shot to attack pigs that are protected by blocks of ice, woods and stones. Since AI agents only get screenshots as input information, the first topic for the problem is computer vision. It is basic and important to lay a foundation for the follow-up works.

The objects in the Angry Birds game can be divided into two categories. Birds, pigs, woods, stones, ice and TNTs are dynamic objects, while slingshot, ground and mountains are static. Dynamic objects can collide with each other and behave according to the law of physics. On the contrary, static objects cannot move or collide with other static objects. When a dynamic object collide with a static object, they follow the law of physics in the way that the static object has infinite mass. In addition to mobility, there is another difference between the two categories – dynamic objects are all convex polygons, while static ones can be concave. This difference leads to different methods of object detection.

Angry Birds Game Playing Software (Version 1.1) [Ge *et al.*, 2013] has realized a vision model to detect and represent dynamic and static objects. It use Axis-aligned minimum bounding box (called AABB) to represent dynamic objects. AABB is a well-known concept in the area of computer vision, but it is not accurate enough because the shapes and angles of some objects, such as woods, stones and ice, can be various. Another weakness of this vision model is that it does not detect static objects other than slingshot. It is necessary to detect and represent other static objects since that ground appears in all levels and mountains often play important roles in the levels they appear.

To improve the performance of representation of dynamic objects, we detect bounding convex polygons (BCPs) from AABBs of objects. BCP can represent shape and angle of an object accurately in theory, while it often misses some bits of the object in practice. To avoid that partly, we apply an algorithm to restore rectangles that are incomplete.

For static object detection, the main task is to detect mountains. As mentioned above, mountains can be concave so that we cannot use the same method to detect them as dynamic objects. We apply an algorithm that contains the following steps. Firstly, we capture a rough shape of mountains through color detection, and then find out the boundary. After that, we apply a Hough Transform to get straight lines that fit the boundary. Finally we segment the straight lines and form polygons to represent mountains.

In the following, we review the previous work, and introduce dynamic object detection and static object detection respectively.

2 Previous Work

Our work is based on the project Angry Birds Game Playing Software (Version 1.1). However, since the vision as well as some other related models is quite simple and inaccurate, we focus a lot on how to improve the performance of these basic models.

The first part we need to improve is how to capturing dynamic objects (including woods, stones, ice and so on) in a more efficient way.

From reading the code of Angry Birds Game Playing Software, we figure out that the original method for finding these objects is:

1. Find an unused point which is of the same color as the main part of the corresponding object. For instance, woods are mainly of color 481, while stones are color 365 and ice is 311.
2. Use FLOOD-FILL algorithm to find out all points which near the start point and may be parts of the object, by checking their colors.

3. Return the AABB of the points set as the whole object. It may be discarded if it is too small.

Using AABB to represent an object is easy for implementation, however, it makes the deeper researches become much more difficult. That is because this representation will lose lots of information about the actual object, such as the shape (Figure 1), the position (Figure 2), and even the relation between two or more connected objects (Figure 3).

Figure 1: Lose shape information of the top three round stones at Level 8, however, the optimal strategy is trying to make them rolling down

Figure 2: The left most and the right most woods at Level 5 look exactly the same, however, far from the truth

3 Dynamic Object Detection

3.1 Applying Bounding Convex Polygon

Obviously, the shapes of dynamic objects - whatever rectangles or circles or triangles - all can be estimated by convex polygons.

In that case, we use the minimum Bounding Convex Polygon (called BCP) of the points set instead of the bounding box of it to represent the object.

Once the points set of an object is captured, we get its bounding convex polygon using the algorithm below:

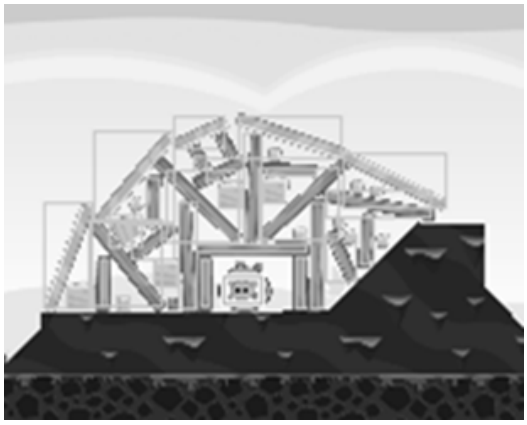

Figure 3: These woods and ices at Level 19 become totally a mess under this representation

1. Let X_L be the x-coordinate value of the leftmost point, and X_R be the x-coordinate value of the rightmost point.
2. For each integer X between X_L and X_R, find the points with the smallest or the largest y-coordinate value. While the points set is generated by FLOOD-FILL algorithm, it is certain that at least one point can be found on each X.
3. Arrange the points with largest y-coordinate for each X from left to right to form the "lower bound" of the object, and arrange those ones with smallest y-coordinate for each X from right to left to form the "upper bound".
4. Scan all points on the bounds in order, use Cross Product [Cormen *et al.*, 2001] to check whether they should be on the final bound, and discard the ones which are not. Then we get the BCP of this object.

Figure 4 may help to understand the whole process.

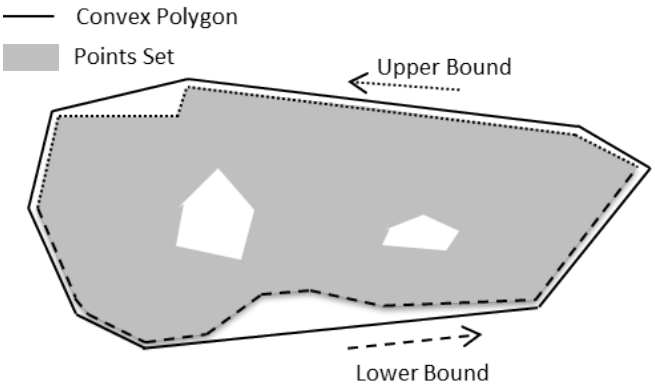

Figure 4: Calculating bounding convex polygon

After applying the BCP for object representation, the problems caused by information lost can be effectively solved (Figure 5).

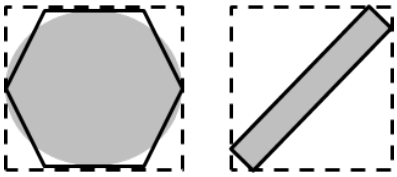

Figure 5: Comparison of bounding boxes (dotted lines) and bounding convex polygons (solid lines)

3.2 Restoring Rectangles

When the BCP representation is applied in our project, it seems that a new problem has been reared up. Since we only use color to check whether a point is belong to an object, the object often miss one or more bits from the corners because it is not always of pure color.

While most of the objects are in rectangles, we are trying to fit the objects we got into rectangles, too. The basic idea is to find a minimum rectangle which is able to contain the corresponding object.

In order to avoid incorrectly converting a circle to a rectangle, we set a constant threshold K, then if the ratio of the area of the BCP to the area of the minimum rectangle is less than K, the original shape of the object is possibly not a rectangle, and vice versa.

The threshold is set to be

$$K = \frac{\pi + 4}{8}$$

which is the average result of a circle ($radio_{circle} = \pi/4$) and a rectangle ($radio_{rectangle} = 1$).

When finding the minimum rectangle containing a convex polygon, we can assume that one side of the rectangle coincides with one side of the convex polygon. Therefore, we enumerate each side of the convex polygon as the bottom side of the rectangle, determine the left side and the right side by using Dot Product, and figure out the height of the rectangle by calculating the distance between each point and the bottom side, using Cross Product [Cormen *et al.*, 2001]. (Figure 6)

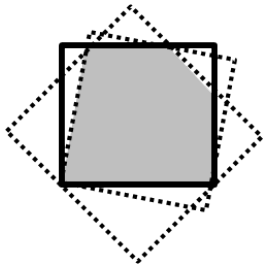

Figure 6: Finding the minimum rectangle

4 Static Object Detection

4.1 Ground Detection

After went through level 1 to 21, we find a good news that the ground in each level is exactly at the same place, which means we can simply use a fixed rectangle to represent it.

4.2 Mountain Detection

Representing the ground is quite easy; however, representing the mountains on the ground is not an easy task. That is because mountains are concave polygons. Thanks to many useful tools (such as edge detection and Hough Transform) from the computer vision literature, we can solve this problem in an efficient way.

To detect mountains, we take the following steps:

1. Find all points on the mountain boundary by edge detection;
2. Find straight lines to fit the mountain boundary by Hough Transform;
3. Use segments to represent the mountain boundary;
4. Use polygons to represent the mountain boundary.

For example, we detect mountains from a screenshot as Figure 7.

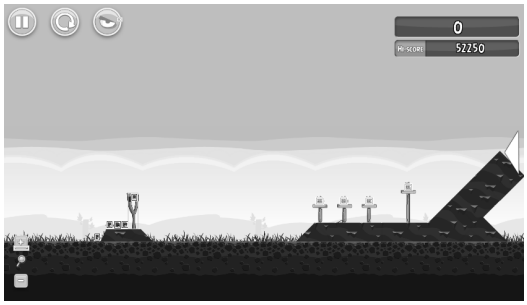

Figure 7: Original image

Boundary Extraction

We find that mountains have their own color which is different from other objects. So it is easy to find all points on the mountain as foreground points, and other points as background points. In Figure 8, the grey points are background points and the white points are foreground ones. We regard a point as a boundary point if and only if it is a foreground point and it is adjacent to a background point. In this way, we can get a mountain boundary represented by a set of points (Figure 9).

Figure 8: Rough shape of mountain

Figure 9: Boundary point extraction

Boundary Line Extraction

We apply Hough Transform to extract boundary lines from the boundary points extracted before.

Hough Transform is a famous feature extraction technique used in image analysis, computer vision, and digital image processing [Wikipedia, 2013].

The main idea of Hough Transform is to consider the characteristics of the straight line in terms of its parameters according to slope-intercept model, instead of image points. For computational reasons, we use polar coordinate model (Figure 10) instead of slope-intercept model.

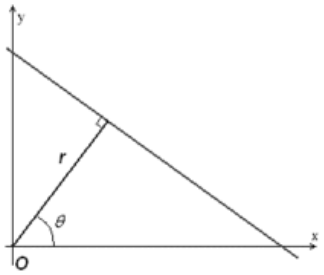

Figure 10: Polar coordinate model in Hough Transfer

The parameter r is the distance from origin to the line, while θ is the angle of the vector from the origin to foot point. With this parameterization, the line can be described as

$$y = \left(-\frac{\cos\theta}{\sin\theta}\right) \cdot x + \left(\frac{r}{\sin\theta}\right)$$

which can be rearranged to

$$r = x\cos\theta + y\sin\theta$$

Therefore it is possible to represent each line with a pair (r, θ) which is unique if $\theta \in [0, \pi)$ and $r \in \mathbb{R}^*$.

After Hough Transform, we get the boundary lines in Figure 11.

Boundary Segment Extraction

After boundary line extraction, we get some lines to fit the boundary points. However, there are still many points on the lines that are not on the boundary. So we need to use segment extraction to get segment lines to form the boundary. We use the GET-SEGMENTS algorithm to complete this task. We consider two boundary points are in different segment lines if between them there are more than five continuous points that are not boundary points. Finally, we get the result as shown in Figure 12.

Figure 11: Boundary line extraction

Figure 12: Boundary segment extraction

GET-SEGMENTS(LINES, BOUNDARY_POINTS)

```
1  segments ← {}
2  for line ∈ lines
3      do cnt_cont ← 0
4         cnt_miss ← 0
5         for p ← each point on line
6             do hit ← false
7                for p' ← each point adjacent to p
8                    do if p' ∈ boundary_points
9                           do hit ← true
10                             break
11               if hit
12                   do if cnt_cont = 0
13                          do seg ← new segment
14                             Set p as the begin of seg
15                      else  Set p as the end of seg
16                      cnt_cont ← cnt_cont + 1
17                      cnt_miss ← 0
18                 else  cnt_miss ← cnt_miss + 1
19               if cnt_miss ≥ 5 and cnt_cont ≥ 5
20                   do Insert seg into segments
21                      cnt_cont ← 0
22                      cnt_miss ← 0
23 return segments
```

In GET-SEGMENT algorithm, we scan each line from one end to the other. For each point on the line, we check whether

Level	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	Avg.
T_{AABB}	72	71	67	70	68	78	69	77	72	71	77	79	71	67	71	75	72	66	70	68	70	71
T_{BCP}	73	87	90	88	83	82	91	77	88	91	91	94	83	87	93	96	92	95	82	91	111	89
Acc%	100	100	86	100	97	100	50	100	79	94	95	63	81	100	94	100	80	80	71	93	80	88

Table 1: Evaluation result of dynamic object detection on the initial state of each level

it is on the boundary from Line 7 to Line 10. If it is (we call it "hit"), then we increase the continuous count (cnt_{cont}) by one and clear the miss count (cnt_{miss}); Otherwise, increase the miss count by one. Once the miss count is greater than five, we believe that the current segment is completed. In order to avoid the negative effects of image noise, only the segments longer than five are considered (see Line 19).

Boundary Polygon Description

The segments we got in the previous work are out-of-order. In that case, we have to sort them in either clockwise or anti-clockwise to form a polygon. We use the GET-POLYGONS algorithm to achieve this goal.

GET-POLYGONS(SEGMENTS)

```
1  polygons ← {}
2  for seg ← unused segment in segments
3       do polygon ← new polygon
4          Add seg into polygon
5          for seg' ← each segment in polygon
6               do for seg'' ∈ segments
7                    if seg'' interacts seg'
8                         do Add seg'' into polygon
9                            Delete seg'' from segments
10         Add polygon into polygons
11 return polygons
```

After all works above are completed, we get the final image in Figure 13.

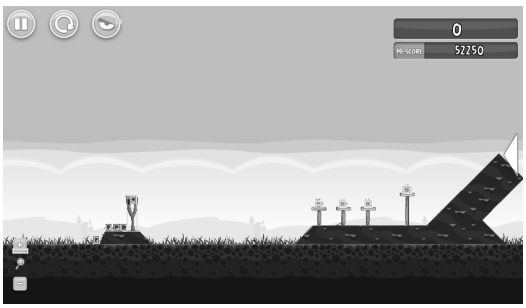

Figure 13: Final image

5 Evaluation

We apply both dynamic object detection method and static object detection method to the real game states, in order to evaluate the actual effect of them. To eliminate the uncertain factors, only the initial state of each level is considered.

5.1 Evaluation Result of Dynamic Object Detection

We first compare the difference about the time cost between the original method – axis-aligned minimum bounding box (AABB), and our new method – bounding convex polygon (BCP). The average time cost of calculating AABB is 71 millisecond, while the average time cost of calculating BCP is 89 millisecond. It turns out that our method is about 20 millisecond slower than the original one on average for each time. However, it is obviously that only before shooting a new bird will dynamic object detection be used. In another word, since there are only five birds in each level on average, our method just need about 100 millisecond more to achieve its goal. The extra time can be ignored.

We also evaluate how accurate our method will be during dynamic object detection. We define the accuracy rate of detecting dynamic objects on a certain state as: the ratio of the number of correctly detected dynamic objects to the total number of dynamic objects on this state. The average accuracy rate is 88%. After analyzing the result carefully, we summarize three main problems that will cause inaccuracy:

- Some objects are too small to recognized, even hard for people.
- There is no clear-cut boundary between two objects, so the program will regard them as the whole thing; this problem often occurs when two pieces of ice touch each other.
- The color of an object changes sharply, then the program may incorrectly divide it into two or more parts; this situation often appears on stones.

Fortunately, all this problems will not cause severe consequences. That is because objects which are too small can hardly effect on game playing, and in most instances, objects that are tightly touched are difficult to be separated, so they can be regarded as a whole thing, and vice versa.

All evaluation results are shown in Table 1[1].

5.2 Evaluation Result of Static Object Detection

We also evaluate the performance of our static object detection method, and list the result in Table 2.

Our method for static object detection is a little bit complex; so that the time cost is quite long – 324 millisecond on average, around four times of which for dynamic object detection. However, while its goal is to detect static object, this process will run only once for each level. Thus it will not slow down the efficiency of the whole algorithm substantially.

This method provides a quite accurate way of static object detecting. Only when there are too many mountains and too many noises will it make mistakes.

[1] All time are measured in millisecond, the same below.

Level	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	Avg.
Time	134	308	321	325	157	624	175	156	172	350	348	371	584	361	728	203	173	199	198	547	378	324
Acc%	N/A	100	100	100	100	64	100	N/A	N/A	100	100	100	50	100	17	17	N/A	100	100	100	100	85

Table 2: Evaluation result of static object detection on the initial state of each level

6 Conclusion

In this paper, we focus on how to represent the objects more accurately in the Angry Birds world. We divide the objects into two categories – dynamic ones and static ones. We use BCPs to represent convex dynamic objects, while applying edge detection and Hough Transform to build concave polygons to represent static objects.

This work improves the performance of object representation in the Angry Birds game. It would benefit the following works of the AI agent, such as structure analysis and physical simulation. Though our method may require up to one second more time than the original method, the difference can be tolerant since the time limit for each level is three minutes or even more.

There are still many things we can improve. For example, two objects may sometimes be detected as one object as well as one object can be detected as two or more objects, which will be harmful when calculating their weight.

References

[Cormen *et al.*, 2001] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms*. MIT Press, Cambridge, Massachusetts, 2001.

[Ge *et al.*, 2013] XiaoYu Ge, Stephen Gould, and Jochen Renz. *Angry Birds Basic Game Playing Software*. Research School of Computer Science, The Australian National University, March 2013.

[Wikipedia, 2013] Wikipedia. Hough transform. http://en.wikipedia.org/wiki/Hough_transform, July 2013.