AR Flappy Bird – Save the plane

Jiaxin Pan[*]
Robotics, Cognition and Intelligence, TUM

Yifan Huo[†]
Robotics, Cognition and Intelligence, TUM.

Manxi Sun[‡]
Data engineering and analysis, TUM.

Abstract

We build a computer game based on augmented reality. The player can control a virtual plane with a specific marker to avoid virtual missiles. The movement of the marker is detected with the webcam. The game is based on OpenCV and OpenGL and includes a timer, start & end interfaces, marker tracker, and collision test. We also improve the marker tracker algorithm for the sake of robustness.

1 Introduction

We set 3 markers to control the game: one for game starts, one for controlling the airplane, and one for the game restarting. Before starting, the game waits for the marker 0x1228. As soon as this marker is detected, the start interface is shown on the screen to start the game. During the game, the player holds another Marker 0x1C44 to control the airplane to avoid the missiles. We set a timer to record the cumulative game time which is shown on the screen in real-time. If the airplane collides with the missiles, the game ends and the end interface is shown on the screen, including the total gaming time. After the game ends, the player can show the marker 0x0B44 to play again. If this marker is not detected within 10 seconds, the program ends automatically.

Here is a short outline of this paper: in section 2, we introduce briefly the creation of the airplane model and the missile map. In section 3, the improvement of the marker tracker algorithm is discussed in detail. In section 4, the algorithm of the collision test for the airplane and missiles is introduced. In sections 5 and 6, we introduce the timer and the start and end interfaces. In section 7, we make a brief summary of our program.

2 Model creation and infinite loop scene

In our game, we need to control the airplane to travel through the map to avoid flying obstacles, which are called missiles here. In order to create the missile model, we modified the *cylinder()*, *circle()* and *cone()* functions in *DrawPrimitives.h* so that we can configure the radius of the circle and the length of the cone and cylinder. Then we use the *GL_TRIANGLE_STIP* method to draw the tail wing of the missile. The creation of the airplane model also uses the same function to draw the airplane with multiple triangular planes. The missile and airplane models are shown in the Fig. 1(a) and Fig. 1(b). Their model drawing functions are named as *Missile()* and *Airplane2()* and saved in the *DrawPrimitives.h*.

In the game scene, all missiles will fly in from the right side of the screen, and fly out from the left after crossing the screen. In the actual program, there are only four missiles in total, flying side by side in pairs. After the missile disappears from the left side of the screen, it will return to the far right side, thereby forming an infinite loop of the scene. And every time missiles returns to the right, a new height for each missile is randomly generated. Therefore, the position of the missile is not fixed, which adds more difficulty to the game.

[*]e-mail: ge43ley@mytum.de
[†]e-mail: ge64lan@mytum.de
[‡]e-mail: manxi.sun@tum.de

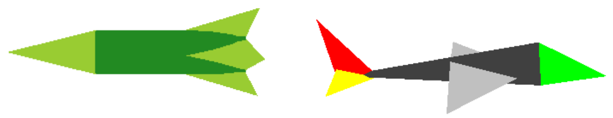

(a) Missile (b) Airplane

Figure 1: Models of the missile and airplane

3 Optimization of marker tracking algorithm

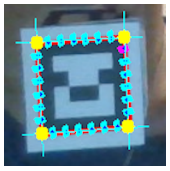
After having the models needed for the game, we need to project the model of the aircraft onto the marker used to control the aircraft. This step is similar to what we did in the tutorial, but the original marker tracking algorithm is very sensitive to light, making the recognition of markers intermittent. When the marker cannot be identified, our aircraft will stay still, which has a great impact on the stability of the game. Therefore, we decided to optimize the marker tracking algorithm.

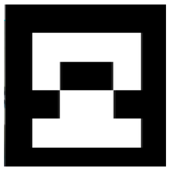
The original algorithm first performs binarization processing on the gray-scale image converted from the original image collected by the camera, and finds the quadrilateral in the black and white image. Afterwards, binarization is performed again on the gray-scale image of the found quadrilateral to identify whether it is a marker. Both two binarizations use a fixed threshold function, which makes it easy for the entire marker to be recognized as pure black or pure white when the light condition changes, such as daytime and night or switching lights. Therefore, here we use the *adaptiveThreshold()* function instead to calculate the average brightness within a certain range set by the *blockSize* parameter and automatically select the appropriate threshold. And we found that when the value is relatively large, the edge of the object can be better recognized; when the value is small, the black and white pixels can be better distinguished. Therefore, we use a larger value of 41 when identifying a marker in the overall image, and a smaller value of 3 when identifying the specific QR code in the marker. The optimized result is shown in theFig. 2.

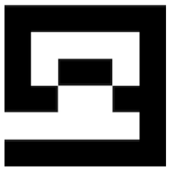
It can be seen from the Fig. 2 that the optimized tracking algorithm is more stable in the face of different light conditions. But at the same time there is a problem, that is, this algorithm will have a certain probability of error when identifying edge pixels. In the original algorithm, the image is recognized as a marker only when the border pixels of the image are all black. Now we relax this condition slightly to require no more than five white pixels among all border pixels. In this way, stability can be ensured, and interference images can be filtered out.

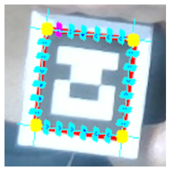
4 Collision test

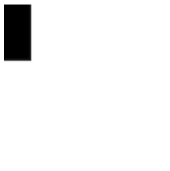
After projecting the aircraft and missiles onto the screen, the next step is the collision test. Because the aircraft moves in 3D space and the missile only moves on the 2D screen, the depth values of the two will not be used as the basis for collision detection, instead, the collision test is only based on their position in the screen coordinate. In this way, we need to calculate the coordinates of the model on the screen. Here we use the *gluProject()* function to achieve this goal. In addition, since the two models are relatively complex polygons, if we project every point onto the screen and then perform very


(a) Lights off


(b) Old


(c) New


(d) Lights on


(e) Old

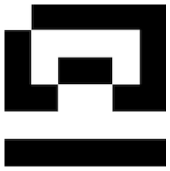
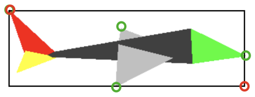
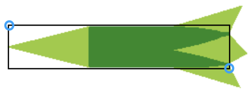
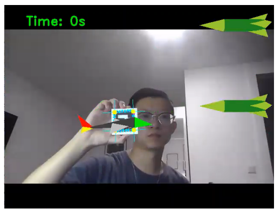

(f) New

Figure 2: Fig. 2(a) and Fig. 2(d) respectively show the marker images collected by the camera when the lights are turned on and off. The original algorithm can identify the marker when the environment is dark, as shown in Fig. 2(b). But after turning on the light, because it uses a fixed threshold, most of the pixels are then recognized as white, as in Fig. 2(e). Our algorithm is highly flexible and can adapt to changes in light.

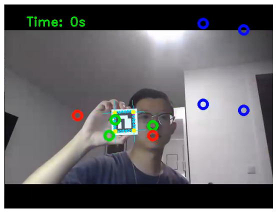
accurate collision detection, it will consume a lot of computing power and make the program unable to run smoothly. Therefore, we choose to select several key points on the aircraft and missile models, and then approximate the model into a quadrilateral according to the position of key points on the screen. The Fig. 3 shows the location of the key points and what they look like in actual operation.


(a) Key points of airplane


(b) Key points of missile



(c) Models

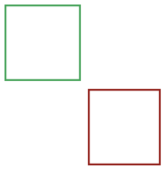


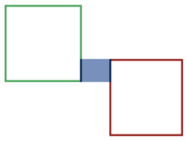
(d) Key points

Figure 3: Key points and approximate rectangle

Fig. 3(a) and Fig. 3(b) respectively show the key points of the aircraft and missile models. Because of the relatively simple motion of the missile, we chose the upper left and lower right corner points as the key points, which is the position shown by the blue circle in Fig. 3(b) and Fig. 3(d). Since the aircraft will rotate in 3D space, we first select four key points on the nose, left and right wings and tail (green circles in Fig. 3(a) and Fig. 3(d)), and then select the maximum and minimum values in the x and y directions to form an approximate rectangle. The red circles in the same image indicate the top left and bottom right corners of the approximated rectangle.

After the aircraft and missiles are approximated by rectangles, we use the axis-aligned bounding box (AABB) [1] algorithm for collision test. We detect whether each side of airplane overlaps with the two sides of each missile that are parallel to this side. If the edges in both the x and y directions overlap, then it will be considered as a collision.


(a) No overlap


(b) One side overlap

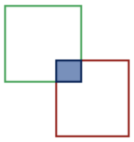

(c) Two sides overlap: collision

Figure 4: Principle of AABB algorithm

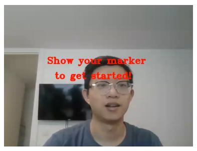
5 Timer

In the whole program, we use 2 timers, one of which is placed on the top left corner within the game to show the time after the game starts, which can be seen in Fig. 3(c), another one is used to wait for playing again.

For the first timer, it is realized with two global time_t variables (by including time.h), namely s_start, which represents the starting point of the game and s_end, recording the current time point. The time period s_start-s_end stands for the gaming time. In the main loop, s_start is set when the marker for starting the game is detected, so it's set only once. On the other hand, the timepoint s_end is recorded in every loop and a variable time_taken is calculated also in every loop after s_end is recorded. The variable $time\_taken = s\_start - s\_end$ is used not only for the timer shown in the top left corner but also for the start interface, which is described in detail in the next section. As for the timer shown in the top left corner, this is realized with the function *timer*. In this function, the variable time_taken is converted to a string and shown on the screen with *cv::putText*. With the timer, the player can intuitively see how long the game has started.

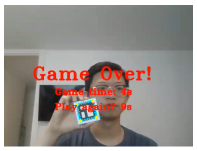
For the second timer, it is active once the system detects a collision and the game is over. Same as the first timer, another two global time_t variables are used, namely e_start and e_end. The timepoint e_start is set only once and it records the time point that the collision is detected. The timepoint e_end is recorded in every loop after the game is over. A new time period is calculated in every loop after the game is over with $e\_end - e\_start$, which is called the waiting time and is used to show how long the system has waited for the marker to play again. This time period is not only recorded in the system but also shown to the player with a countdown, which is described in detail in the next section. If the waiting time is more than the maximum waiting time, which is set to 10 seconds in the system, the program is ended automatically.

6 The Start & The End Interface



(a) Before start

(b) Game start



(c) Game over

Figure 5: The start and end interfaces

The Start & The End interface can be seen as 3 parts depending on time, one before the beginning marker detected, one after the beginning marker detected and after the game starts, and one after game overs.

Before the game is started, a sentence is put on the middle of the screen with *cv::putText*, saying ”show your marker to get started”, which can be seen in Fig. 5(a). This sentence is on the screen until the beginning marker is detected and then it is replaced with other text. This is realized in the function *wait*, which is active before the game starts. In this function, only the background video captured with the camera is shown, no airplane or map, and the beginning marker is detected.

After the game starts, the texts ”3”, ”2”, ”1”, and ”Game Start!” are put on the screen, which is shown in Fig. 5(b). Unlike the sentence before the game starts, these texts are not put together and are shown one after another. This is realized with the timer. After the beginning marker is detected, the time point s_start is recorded. The time period after the marker is detected is calculated in every loop with the variable time_taken. With the help of this variable, the texts are put on the screen, one text for one second. This is realized in the function *textandtimer*.

After a collision is detected, the game gets ended. From this time point, three sentences are put on the screen, one with ”Game Over!”, one showing how long the player has played the game and the other is a countdown, waiting for the player to show the marker to play again. This is shown in Fig. 5(c). The first and sentence stays on the screen and are not changed within the waiting time for playing again. The second sentence shows the playing time with the help of the variable time_taken. The third sentence is changing during the waiting time like a countdown. This is realized with the second timer, which records the time after the game ends. These three sentences are put on the screen also with *cv::putText* in the function *textandtimer*.

7 Conclusion

In this project, we build an AR computer game based on OpenCV and OpenGL. The game combines the movements of the player with a virtual airplane via a marker. A virtual missile map is created randomly in real-time and runs faster and faster, which the airplane needs to avoid. For the sake of completeness, we add start and end interfaces and a timer recording the gaming time. We detect the collision between the airplane and missiles by approximating the models into quadrilaterals according to the positions of the key points. What's more, we improve the marker tracker algorithm to make it more robust to different lightnings.

Here are some possible future works:

- The model of the airplane is not so professional compare with other computer games. We could find other models or try other ways to build models.
- There are blue margins when turning into full-screen mode, further modification to this issue is meaningful.
- In the part of collision detection, both the scope of the airplane and the objects are regarded as rectangles which leads to inaccuracy occasionally. More geometry models or other methods could be considered to represent the scope.

References

[1] J. de Vries. Collision detection. Website, 2014. https://learnopengl.com/In-Practice/2D-Game/Collisions/Collision-detection.