Extracted from:

Augmented Reality

A Practical Guide

Chapter 9

Enhancing Your AR Game

This chapter builds upon the game you started to build in the previous chapter. Now that you have the basics of your game, it's time to add some artificial intelligence (AI) and complete the tank behaviors. The AI of a game is often the most challenging part; even simple behaviors require careful thought and often require a great deal of code. But it's also true that the element that distinguishes one game as more fun and intriguing to play from another is often the quality of the AI. It's extremely difficult to program AI that is as interesting as human opponents. This book is about AR, not AI, so you will create a simple AI system.

9.1 AI for Robot Tanks

An integral part of many video games is some quasi-intelligent behavior of various game elements—usually the bad guys. This sort of AI ranges from the ghosts in Mrs. Gobble to the worker and army units in real-time strategy (RTS) games. Often, as in your tank game, the robot elements need only a minimal level of intelligence. In others, such as an RTS—or in the extreme case, a chess game—the AI is what makes the game.

Regardless of how involved the AI is, it needs to be well thought out. Without sufficient planning, the robot behavior can turn out to be far different from what you expected. Also, poor planning can lead to many ad hoc changes and "repairs" that make your code undesirable. Not to suck the fun out of programming a video game, but game AI quickly becomes a complex state machine, and too much interactivity between behaviors can lead to complex feedback paths.

Trying to fix one thing can end up causing something else in turn to act strangely, and after a while, the programmer has to admit that she does not understand the system she has created.

It may sound exciting to have unexpected robot behavior, but unexpected behavior usually means that the robot tanks just sit around, clump together in traffic jams, or inexplicably drive through walls. Unexpected robot behavior usually means a frustrating programming experience—not the robots becoming self-aware.

> "A problem clearly stated is a problem half solved."
> —Dorothea Brande

The more clearly you describe what you want your AI game elements to do, the greater the chance you will end up making a system that works. You should first plan things from a system's viewpoint and then make concrete plans for various parts. Obviously, you should do this before you start coding. If there are any contradictions or undesired outcomes, you'll have to face them at some point, and it's usually easier to do so before you start coding.

There are many ways to plan your game AI behavior. Similar to a chess game, you could examine a number of possible next moves and then for each consider several different behaviors. Then, for each option, examine a number of possible moves. This expands in an exponentially growing tree of possibilities, so it is more suited to a strategy game than your tank game.

An alternate AI system architecture is to design a set of behaviors the agent (tank) can be doing one at a time. Since only one behavior can be acted upon at a time, these behaviors are prioritized. A behavior is activated by some conditions—if active, each behavior suppresses behaviors lower on the priority list. Also, the lower priority behaviors should somehow move the system to a state where the higher-priority behaviors kick in. Imagine a simplified animal behavior: avoiding a predator is an overriding behavior to finding food and water, which itself is an overriding behavior to finding some entertainment. For example, a wombat's boredom behavior is quickly replaced by predator avoidance behaviors if a hungry fox suddenly appears.

You're going to adopt this architecture of prioritized behaviors in your Tank Wars game. Now you have an architecture framework. The next step is to design a set of behaviors and prioritize them.

Let's clarify how your tanks should behave. They should have the ability to *see* opponent tanks and should try to fire shells at them. The tanks should not be able to see through walls, but they should be able to distinguish friend from foe. Since the missiles don't change trajectory, the tanks need to be able to turn and face the enemy before shooting. Based on these points, let's set down some behavior rules and their priority:

Behavior #1: If a robot tank sees an enemy tank and is lined up with it, it should launch a missile.

Behavior #2: If a robot tank sees an enemy tank, it should rotate to line up with it.

Behavior #1 overrides behavior #2. The easy way to implement priority is to set up an "if, else if, else" structure with the higher (lower priority #) behaviors higher in the "if, else if, else" code. Note that you should be careful what priority you assign to each behavior. For example, if you put the rotate behavior first, then even if a tank had an enemy lined up, it may rotate away to face another. Putting the firing behavior first is logical—if the tank can take a shot, it should fire.

If you give your tanks a limited field of view—as in your game—a robot may not see an enemy tank that is aiming in its direction. In this case, the robot tank should do something when it gets hit by an incoming missile; this is behavior #3.

Behavior #3: If a robot tank gets hit by a missile, it will rotate to face the direction of the oncoming missile.

These three behaviors act in a chain reaction from low to high priority. If a tank is minding its own business and a missile suddenly hits it, behavior #3 activates, and it turns to face the threat; perhaps it should run away, but that's more complex. Once the tank rotates far enough to see the enemy tank that fired, behavior #2 kicks in, and it keeps rotating to line up with the aggressor (which may be a different angle, if the enemy has moved since it fired the original missile). Once a tank has its turret lined up with the enemy, behavior #1 activates, and it fires. The first few stages in the next section will help you implement these three behaviors.

You can simplify the situation by having the first three behaviors happen when the tank is not moving, so let's declare that the tank will brake if it's moving and wants to perform behavior #1, #2, or #3.

So far, you have basically added the behavior of a gun turret, so now you will add some motion. What should a tank do if not immediately in a battle (behaviors #1–#3)? A logical behavior would be to go help another tank on its team, but how do you define help? Remember that you don't want to directly battle the enemies with his lower-priority behavior; you only want this behavior to bring about a situation that invokes the higher-priority behaviors—the ones responsible for battle. With this in mind, let's define helping a fellow tank that is in a battle:

Behavior D: A robot tank will drive to the (X, Y) coordinates of a team member that currently sees an enemy.

Note the switch from numbered behaviors to letters; you're going to renumber the ones with letters later in this chapter. Also note that you don't need to preface the behavior rule by "If a robot is just sitting around doing nothing, then..." because this is implicit in our priority architecture.

You can imagine that these behaviors could still result in some strange results. The default behavior—if conditions don't trigger behaviors #1 to #3 or D—is to sit still. This means the game can get stuck in a state where all tanks are just sitting still in different parts of the map. Therefore, you will add patrolling as the final behavior. If none of the tanks on a team is in battle, then they'll all drive around until one finds an enemy. This will trigger behavior D, and the rest of the tanks will roll over.

Patrolling, like *helping*, is a vague concept, so you need to clarify it into a simple and precise behavior. In Tank Wars, you'll implement searching by picking a semirandom destination in the map and driving toward it. Specifically, you'll have a list of *waypoints* for each team, and each "bored" tank will pull a waypoint off the list and navigate to it. The next bored tank will pull the next waypoint off the list. The list of waypoints will be chosen so that the tanks fan out; this will stop them from all driving to the same location.

Behavior E: A robot tank will pick a waypoint off a list—according to a variable specific to the robot's team—and will navigate to the waypoint. The variable will then be incremented so that the next bored robot picks a different destination.

Note that behaviors D and E both involve navigating around the map while avoiding walls. Thus, you need some path planning. This will cause you to subdivide these behaviors, but let's leave that out for now.

You now have a behavior priority tree with five behaviors identified (behaviors #1, #2, #3, D, and E). Next, you'll see behaviors #1, #2, and #3 implemented, and then you'll address the path planning for behaviors D and E.

9.2 Vision for Robot Tanks

If a robot tank can see an enemy, behaviors #1 and #2 require processing to determine the angle of the enemy tank. You want to have some realism in that a tank cannot see through the walls, so you'll add some tunnel vision to give them a limited field of view; this will allow other tanks to sneak up on them.

How are you going to achieve this robotic tank vision? You aren't going to try a full simulation and render an image from the tank's viewpoint and process this image for enemy tanks. However, this would be a good project for a computer vision researcher and may not be intractable if you identify the presence of tanks by color. Since you already have all the data for the robot tanks' positions, you can cut some corners and simulate the same effect.

In this section, you're not going to implement any behaviors; you're just going to add the *vision system*. Let's start by breaking down the vision system into more precise terms.

A robot tank A sees an enemy tank B if B's location in polar coordinates —centered on A—is within Θ of the direction A is facing (where Θ is half the field of view of tank A) and where there are no wall elements on the line segment between tank A and tank B.

The vision system is going to provide two outputs: a Boolean flag enemy_seen and a floating-point relative angle enemy_delta_angle. The angle tells you how many degrees to turn to face the enemy. The angle enemy_delta_angle is positive if the enemy tank is located clockwise from the friendly tank. Remember that you have defined clockwise as positive. If multiple enemy tanks are visible, enemy_delta_angle is the smallest angle of the angle to all enemies. After all, you want the tank to try to fire at the one it can hit first. If this angle is below a threshold, then the Boolean variable enemy_lined_up will be true; this tells you it's OK to fire a shell.

Here are the DEFINE statements related to tank properties. You're adding the ones for visibility:

Download YourARGame/Tank Wars Projects/twars_7_vision/twars_7_vision.cpp

```
#define SHOTS_PER_TANK 3
#define MISSILE_SPEED 8.0
//give tanks 120 degree field of view in front
#define TANK_FOV   120
#define HALF_TANK_FOV  TANK_FOV/2.0
//shoot if enemy is with 5 degrees of straight ahead
#define TANK_ALIGNED   5
```

Here are the new elements in the tank_s structure:

Download YourARGame/Tank Wars Projects/twars_7_vision/twars_7_vision.cpp

```
struct tank_s
{
   //current position
   float x,y,z,angle;
   //last position
   float last_x,last_y,last_z,last_angle;
   //translational and rotational velocity
   float velocity,dangle;
   //reset all movement requests
   bool move_cw,move_ccw,move_fast,move_slow,move_fire;
   //physics setting for this tank
   float max_rot_rate,max_speed;
   //status and actions
   //true=AI controlled, false=user controlled
   bool robot;
   int team;
   int damage;
   //true=active, false=destroyed
   bool status;
   //true for one time tick after collision with wall or missile
   bool crash;
   //missile info
   bool request_fire;
   //time in frames since tank last fired
   int reload_time;
   int time_to_reload;  //reload time setting
   //info on incoming missiles that recently hit tank
   bool recently_hit;  //true if tank has been hit
   float hit_angle;
   //vision system input
   bool enemy_seen;
   float enemy_delta_angle;
   //enemy tank is lined up, ready to fire a missile
   bool enemy_lined_up;
};
struct tank_s tank[NUM_TANKS];
```

Before you forget, initialize these booleans to false so that when you institute behaviors #1 and #2, the robot tanks don't misfire a missile as the game begins. As a conscientious tank commander, you want to avoid collateral damage.

Download YourARGame/Tank Wars Projects/twars_7_vision/twars_7_vision.cpp

```
for(int tank_num=0;tank_num<NUM_TANKS;tank_num++)  {
      tank[tank_num].dangle=0.0;
      //tank "performance" settings
      tank[tank_num].max_rot_rate=15.0; tank[tank_num].max_speed=5.0;
      //initial status and action settings
      //set all tanks to manual control
      tank[tank_num].robot=false;
      tank[tank_num].team=tank_num/5; tank[tank_num].status=true;
      tank[tank_num].crash=false; tank[tank_num].move_fire=false;
      tank[tank_num].damage=10;
      //reset all movement requests
      tank[tank_num].move_cw=false; tank[tank_num].move_ccw=false;
      tank[tank_num].move_fast=false; tank[tank_num].move_slow=false;
      tank[tank_num].request_fire=false;
      //missile related struct elements
      //time in frames since tank last fired
      tank[tank_num].reload_time=0;
      //reload time setting
      tank[tank_num].time_to_reload=4;
      //indicates if tank has recently been hit
      tank[tank_num].recently_hit=false;
      //robot tank AI
      tank[tank_num].enemy_seen=false;  tank[tank_num].enemy_lined_up=false;
   }
```

You're going to put all the AI code into a new function called game_engine_ai(), which you will call before game_engine_physics(). This new function is called from opengl_tick(); here is a code fragment with a single new line:

Download YourARGame/Tank Wars Projects/twars_7_vision/twars_7_vision.cpp

```
//update game
game_engine_ai();
game_engine_physics();

glutPostRedisplay();
```

And finally, here is the vision system implemented in game_engine_ai(). A double nested loop goes through all combinations of tanks. The outer loop goes through values of tank_num_a where A is the tank you're currently processing the vision for, and the inner loop goes through values of tank_num_b where B is the (potential) enemy tank.

Download YourARGame/Tank Wars Projects/twars_7_vision/twars_7_vision.cpp

```
void game_engine_ai(void)
{
//vision for tanks
for(int tank_num_a=0;tank_num_a<NUM_TANKS;tank_num_a++)
   if(tank[tank_num_a].status)  {
     tank[tank_num_a].enemy_seen=false;  tank[tank_num_a].enemy_lined_up=false;
     tank[tank_num_a].enemy_delta_angle=HALF_TANK_FOV;
     for(int tank_num_b=0;tank_num_b<NUM_TANKS;tank_num_b++)
       if( (tank_num_a!=tank_num_b)&&(tank[tank_num_a].team!=tank[tank_num_b].team)
         //only consider alive tanks on different team
         &&(tank[tank_num_b].status) )  {
         float dx=tank[tank_num_b].x-tank[tank_num_a].x;
         float dy=tank[tank_num_b].y-tank[tank_num_a].y;
         float heading=get_angle(dx,dy);  //angle of B w.r.t A
         float heading_diff=heading-tank[tank_num_a].angle;
         if(heading_diff>180.0) heading_diff-=360.0;
         if(heading_diff<-180.0) heading_diff+=360.0;
         if(fabs(heading_diff)<HALF_TANK_FOV)  {
         //an enemy tank B does satisfy the angle criteria
         //see if tank B is in line of sight
         bool line_blocked=check_line_of_sight(tank[tank_num_a].x,tank[tank_num_a].y,
             tank[tank_num_b].x,tank[tank_num_b].y);
         if( (fabs(heading_diff)<fabs(tank[tank_num_a].enemy_delta_angle))
           &&(line_blocked==false))  {
           //enemy tank is visible, and is closest in angle so far
           //of all tanks checked
           tank[tank_num_a].enemy_delta_angle=heading_diff;
           tank[tank_num_a].enemy_seen=true;
           //see if tank B is within sights to fire missile
           if(fabs(heading_diff)<TANK_ALIGNED)
             tank[tank_num_a].enemy_lined_up=true;
             }
           }
         }
      }
}
```

In this code, you used a function called check_line_of_sight(x0, y0, x1, y1) to determine whether the line segment between tank A (assumed to be at x0, y0) and B (assumed to be at x1, y1) crosses a map square containing a wall.

This was put into a separate function because you're going to need this functionality later in your path planning.

Download YourARGame/Tank Wars Projects/twars_7_vision/twars_7_vision.cpp

```
//see if a line segment crossed a "wall" square in the map
//check_line_of_sight() returns true if line segment does cross wall
bool check_line_of_sight(float x0, float y0, float x1, float y1)
{
   float dx=x1-x0;
   float dy=y1-y0;
   float distance=(float)sqrt(dx*dx+dy*dy);
   bool obstacle_in_way=false;
   float step_x=5.0*dx/distance,step_y=5.0*dy/distance;
   //send a text particle along trajectory to see if
   //it lands in a wall square
   float test_x=x0;
   float test_y=y0;
   for(float d=0.0;(d<=distance)&&(obstacle_in_way==false);d+=5.0)
   {
   int xm,ym;      //map coordinates
   //convert from world coordinates (in tank struct)
   //to map coordinates
   //using Eqn. 2 in chapter 9
   xm=(int)(0.1*test_x);
   ym=(int)(0.1*test_y);
   //check map image for wall square
   if(texmap[(xm+ym*MAP_WIDTH)*3]>50)
    obstacle_in_way=true;
   test_x+=step_x; test_y+=step_y;
   }
   return obstacle_in_way;
}
```

Since you are not yet implementing the behavior rules, you need some way to see whether your visibility code is working. To perform the test, you'll simply enlarge a tank depending on the Boolean flags enemy_seen and enemy_lined_up. If the former is true, the tank will be enlarged to 150%, and if the latter is true, it will be enlarged by 200%. Here is the code to do this temporary visualization:

Download YourARGame/Tank Wars Projects/twars_7_vision/twars_7_vision.cpp

```
//--------adjust tank pose and render--------
//save modelview matrix for next iteration
glPushMatrix();
//convert from world coords to marker coords.
//Eqn. 1 in chapter 9
xa=0.5*tank[tank_num].x-80.0; ya=0.5*tank[tank_num].y-80.0;
glTranslatef(xa,ya,0.0);
glRotatef(tank[tank_num].angle,0,0,1);
```

```
//start temporary code for twars_7_vision to
//demonstrate vision engine
if(tank[tank_num].enemy_lined_up) glScalef(2.0,2.0,2.0);
else if(tank[tank_num].enemy_seen) glScalef(1.5,1.5,1.5);
//end temporary code for twars_7_vision to
//demonstrate vision engine

meshman_render_opengl(model_num,1,0);
glPopMatrix();
                }
```

Figure 9.1, on the next page, shows the green tank enlarging to show that it has detected an enemy tank and is lined up with it (that is, ready to fire a missile). This causes the green tank to change from 100% to 150% and then to 200% of its normal size. The green tank starts at 100% in the top image; in the middle image, an enemy blue tank is in sight but not lined up, so the green tank is 150% of its normal size. In the bottom image, an enemy tank is visible and it's in the green tank's gun sights, so the green tank is 200% of its original size.

Here is the function get_angle(). You called this function in game_engine_ai() to turn an X and Y position into an angle (in degrees). This functionality will be needed several more times in the coming sections. As with other functions, the function is put at the top to avoid a function prototype.

Download YourARGame/Tank Wars Projects/twars_7_vision/twars_7_vision.cpp

```
float get_angle(double x,double y)
{
if(x!=0.0)
     {
     if(y>0.0) return atan2(y,x)*180.0/M_PI;
     else      return 360.0 + atan2(y,x)*180.0/M_PI;
     }
else
     {
     if(y<0.0) return 270.0;
     else      return 90.0;
     }
}
```

Now that you know it works, you'll remove the expanding tank visualization code in the next step.

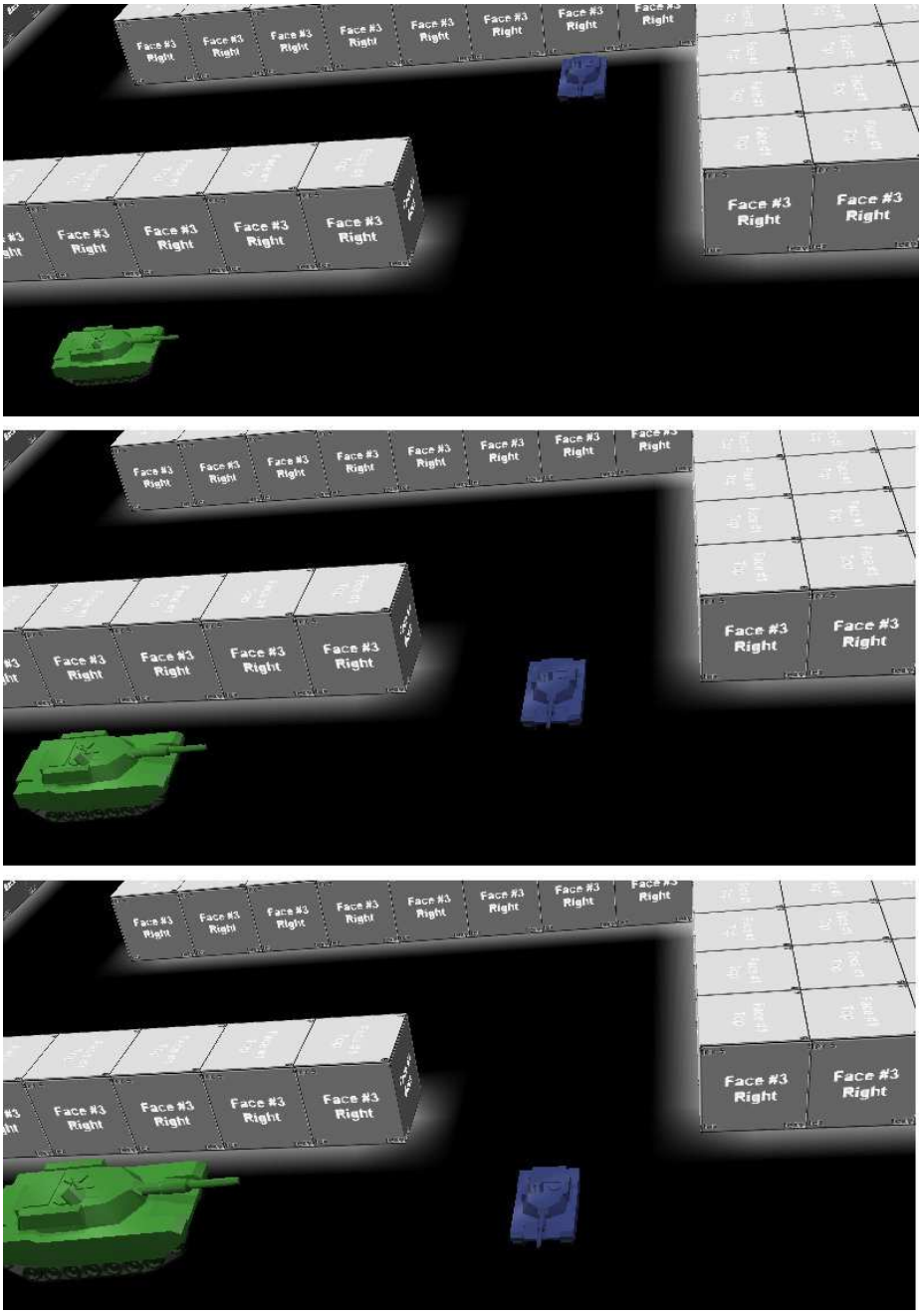

Figure 9.1: The green tank changes size as it detects an enemy blue tank.

The Pragmatic Bookshelf

The Pragmatic Bookshelf features books written by developers for developers. The titles continue the well-known Pragmatic Programmer style and continue to garner awards and rave reviews. As development gets more and more difficult, the Pragmatic Programmers will be there with more titles and products to help you stay on top of your game.

Visit Us Online

Augmented Reality's Home Page
http://pragprog.com/titles/cfar
Source code from this book, errata, and other resources. Come give us feedback, too!

Register for Updates
http://pragprog.com/updates
Be notified when updates and new books become available.

Join the Community
http://pragprog.com/community
Read our weblogs, join our online discussions, participate in our mailing list, interact with our wiki, and benefit from the experience of other Pragmatic Programmers.

New and Noteworthy
http://pragprog.com/news
Check out the latest pragmatic developments in the news.

Buy the Book

If you liked this PDF, perhaps you'd like to have a paper copy of the book. It's available for purchase at our store: pragprog.com/titles/cfar.

Contact Us

Phone Orders:	1-800-699-PROG (+1 919 847 3884)
Online Orders:	www.pragprog.com/catalog
Customer Service:	orders@pragprog.com
Non-English Versions:	translations@pragprog.com
Pragmatic Teaching:	academic@pragprog.com
Author Proposals:	proposals@pragprog.com