# 9

# USING EXTRASENSORY PERCEPTION TO WARD OFF FOG OF WAR

*Fog of war* (often shortened to just *fog*) is a mechanism that game developers commonly use to limit a player's situational awareness and hide information about the game environment. Fog is often a literal lack of sight in massive online battle arena (MOBA) games, but the concept also includes any lack or obscurity of pertinent gameplay information. Cloaked figures, dark rooms, and enemies hiding behind walls are all forms of fog.

Game hackers can reduce or even completely remove fog using an *extrasensory perception (ESP)* hack. An ESP hack uses hooking, memory manipulation, or both to force a game to display hidden information. These hacks take advantage of the fact that some types of fog are often implemented on the client side, as opposed to the server side, meaning that the game clients still contain information (partial or complete) about what is being hidden.

In this chapter, you will learn how to implement different types of ESP hacks. First, you'll learn to light up dark environments. Next, you'll use x-ray vision to see through walls. Finally, you'll learn about zoom hacking, tweaking heads-up displays, and other simple ESP hacks that can reveal all sorts of useful (but otherwise hidden) information about the game you're playing.

## Background Knowledge

This chapter starts the transition from hacking, puppeteering, and reverse engineering to coding. From here on out, you'll be learning how to actually code your own hacks. To keep on topic, everything I've talked about thus far will be treated as background knowledge. If you see a technique used that you don't quite remember, such as memory scanning, setting memory breakpoints, hooking, or writing memory, flip back to the relevant chapters and study them a bit more before continuing. Throughout the text, you'll find notes to remind you where you can brush up on certain topics.

Specifically, this chapter will talk a lot about Direct3D. In "Applying Jump Hooks and VF Hooks to Direct3D" on page 175, I explained how to hook into a game's Direct3D drawing loop. The example code for that chapter includes a fully featured Direct3D hooking engine in *GameHackingExamples/Chapter8_Direct3DHook*. A lot of the hacks in this chapter build on that hook, and their example code can be found in the *main.cpp* file of the Direct3D hook code. You can run the compiled application from *GameHackingExamples/Chapter8_Direct3DApplication* to see the hacks in action on a test application.

## Revealing Hidden Details with Lighthacks

*Lighthacks* increase lighting in dark environments, allowing you to clearly see enemies, treasure chests, pathways, and anything else that is normally obscured by darkness. Lighting is often a cosmetic change that's added at a game's graphical layer, and it can usually be directly modified with a hook on the graphics layer.

Optimal lighting depends on camera orientation, environment layout, and even specific traits of a game's engine, and you can manipulate any of these factors to create lighthacks. But the easiest way is simply to add more light to a room.

### Adding a Central Ambient Light Source

The online resources for this book include two small lighthack examples. The first is the `enableLightHackDirectional()` function in *main.cpp*, which is shown in Listing 9-1.

```
void enableLightHackDirectional(LPDIRECT3DDEVICE9 pDevice)
{
    D3DLIGHT9 light;
```

```
    ZeroMemory(&light, sizeof(light));
    light.Type = D3DLIGHT_DIRECTIONAL;
    light.Diffuse = D3DXCOLOR(0.5f, 0.5f, 0.5f, 1.0f);
    light.Direction = D3DXVECTOR3(-1.0f, -0.5f, -1.0f);

    pDevice->SetLight(0, &light);
    pDevice->LightEnable(0, TRUE);
}
```

*Listing 9-1: A directional lighthack*

This code is called from the `EndScene()` hook, and it adds light to the scene by creating a light source called `light`. The code sets `light.Type` to directional, which means the light source will act like a spotlight and project light in a specific direction. The code then sets the red, green, and blue values of `light.Diffuse` to 0.5, 0.5, and 0.5, giving the light an off-white shine when reflected from a surface. Next, it sets `light.Direction` to an arbitrary point in the three-dimensional space. Finally, the code uses the game's Direct3D device to set up the light at index 0 and enable lighting effects.

**NOTE** *In the example application, the light shines up and to the right from the bottom left of the scene. You may need to change this location depending on how your target game is rendered.*

Note that inserting the light at index 0 works for this proof of concept, but it won't always work. Games typically have multiple light sources defined, and setting your light at an index the game uses might override critical lighting effects. In practice, you might try setting the index to an arbitrarily high number. There's an issue with this type of lighthack, though: directional lights will be blocked by objects such as walls, creatures, and terrain, meaning shadows can still be cast. Directional lights work great for wide-open spaces, but not so well for tightly wound corridors or underground caves.

### Increasing the Absolute Ambient Light

The other lighthack method, seen in the `enableLightHackAmbient()` function, is far more aggressive than the one in Listing 9-1. It affects the light level globally, rather than adding an extra light source. Here's what the code looks like:

```
void enableLightHackAmbient(LPDIRECT3DDEVICE9 pDevice)
{
    pDevice->SetRenderState(D3DRS_AMBIENT, D3DCOLOR_XRGB(100, 100, 100));
}
```

This lighthack sets the absolute ambient light (which you indicate by passing `D3DRS_AMBIENT` to the `SetRenderState()` function) to a medium-strength white. The `D3DCOLOR_XRGB` macro sets that strength, taking `100` as

its parameters for the red, green, and blue levels. This lights up objects using an omnidirectional white light, effectively revealing everything at the cost of shadows and other lighting-based details.

### *Creating Other Types of Lighthacks*

There are many other ways to create lighthacks, but they differ from game to game. One creative way to affect the light in a game is to NOP the code that the game uses to call the `device->SetRenderState()` function. Since this function is used to set up the global ambient light strength, disabling calls to it leaves Direct3D at the default light settings and makes everything visible. This is perhaps the most powerful type of lighthack, but it requires your bot to know the address of the lighting code to NOP.

There are also memory-based lighthacks. In some games, players and creatures emit light of different colors and strengths, often depending on attributes like their equipment, mount, or active spells. If you understand the structure of the game's creature list, you can directly modify the values that determine a creature's light level.

For instance, imagine a game in which characters emit a bluish ball of light when under a healing or strengthening spell. Somewhere in the game's memory are values associated with each creature that tell the game the color and intensity of light the creature should emit. If you can locate these values in memory, you can change them so that the creatures effectively emit orbs of light. This type of lighthack is commonly used in games with a 2D top-down style, since the orbs around individual creatures produce a cool artistic effect while shedding light on important parts of the screen. In 3D games, however, this sort of hack just turns creatures into blobs of light that run around.

You can also hook the `SetLight()` member function at index 51 in the VF table of the game's Direct3D device. Then, whenever your hook callback is invoked, you can modify the properties of the intercepted `D3DLIGHT9` light structure before passing it to the original function. You might, for instance, change all lights to the `D3DLIGHT_POINT` type, causing any existing light sources in the game to radiate light in every direction like a light bulb. This type of lighthack is very powerful and accurate, but it can produce some disturbing visuals. It also tends to break in any environment that has no lighting, and opaque obstacles still block point light sources.

Lighthacks are very powerful, but they don't reveal anything. If information is hidden behind an obstacle, rather than by darkness, you'll need a wallhack to reveal it.

## Revealing Sneaky Enemies with Wallhacks

You can use *wallhacks* to show enemies that are hidden by walls, floors, and other obstacles. There are a few ways to create these hacks, but the most common method takes advantage of a type of rendering known as *z-buffering*.

### Rendering with Z-Buffering

Most graphics engines, including Direct3D, support z-buffering, which is a way to make sure that when there are overlapping objects in a scene, only the top object is drawn. Z-buffering works by "drawing" the scene to a two-dimensional array that describes how close the object at each pixel on the screen is to the viewer. Think of the array's indices as axes: they correspond to the x-axis (right and left) and y-axis (up and down) for each pixel on the screen. Each value stored in the array is the z-axis value for a pixel.

When a new object appears, whether it is actually drawn on the screen is decided by the z-buffer array. If the spot at the object's x- and y-position is already filled in the array, that means there's another object at that pixel on the screen. The new object will appear only if it has a lower z-axis value (that is, if it's closer to the viewer) than the pixel already there. When the scene is finished being drawn to the array, it is flushed to the screen.

To illustrate this, imagine a three-dimensional space that needs to be drawn to a two-dimensional canvas by some game with 4×4-pixel viewport. The z-buffer for this scenario would look like Figure 9-1.

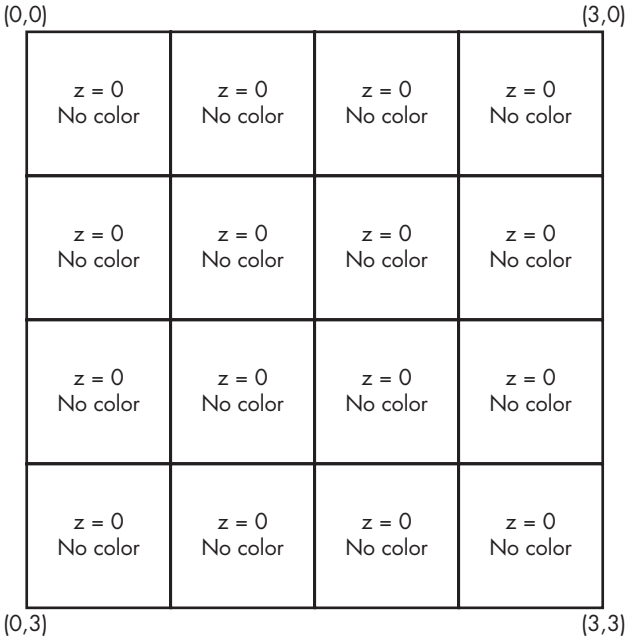


*Figure 9-1: An empty z-buffer*

To start, the game draws a blue background that completely fills the viewport and is located as far away on the z-axis as possible; let's say the highest z-value is 100. Next, the game draws a 2×2-pixel red rectangle at

position (0,0) with a z-position of 5. Finally, the game draws a 2×2-pixel green rectangle at position (1,1) with a z-position of 3. The z-buffer would now look like Figure 9-2.

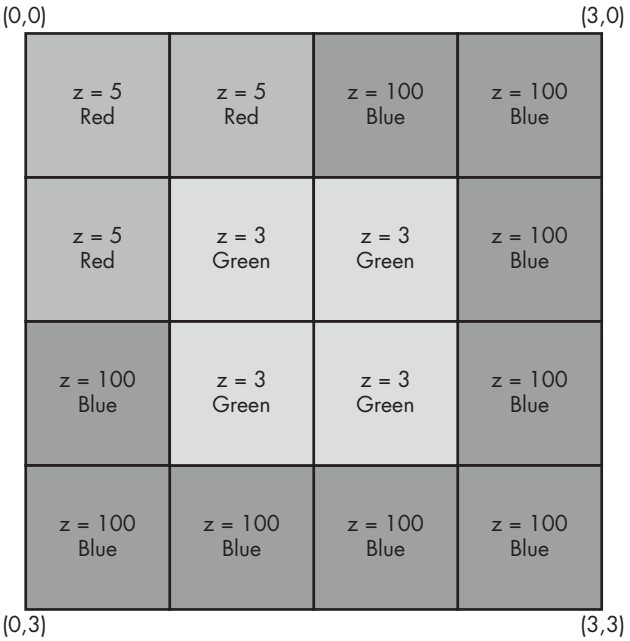


*Figure 9-2: A filled z-buffer*

The z-buffer neatly handled overlapping objects based on their z-positions. The green square that's closest on the z-axis overlaps the red square that's a bit farther away, and both squares overlap the blue background, which is very far away.

This behavior allows a game to draw its map, players, creatures, details, and particles without worrying about what is actually visible to the player. This is a huge optimization for game developers, but it exposes a large area of attack. Since all game models are *always* given to the graphics engine, you can use hooks to detect objects that the player can't actually see.

## Creating a Direct3D Wallhack

You can create wallhacks that manipulate z-buffering in Direct3D using a hook on the `DrawIndexedPrimitive()` function, which is called when a game draws a 3D model to the screen. When an enemy player model is drawn, a wallhack of this type disables z-buffering, calls the original function to draw the model, and then reenables z-buffering. This causes the enemy model to be drawn on top of everything else in the scene, regardless of what's in front of it. Some wallhacks can also render specific models in a solid color, such as red for enemies and green for allies.

### Toggling Z-Buffering

The Direct3D hook in *main.cpp* from *GameHackingExamples/Chapter8_Direct3DHook* has this example wallhack in the `onDrawIndexedPrimitive()` function:

```
void onDrawIndexedPrimitive(
    DirectXHook* hook,
    LPDIRECT3DDEVICE9 device,
    D3DPRIMITIVETYPE primType,
    INT baseVertexIndex, UINT minVertexIndex,
    UINT numVertices, UINT startIndex, UINT primCount)
{
    if (numVertices == 24 && primCount == 12) {
        // it's an enemy, do the wallhack
    }
}
```

This function is used as a callback for a hook on `DrawIndexedPrimitive()` at VF index 82 of the game's Direct3D device. Every model the game draws passes through this function, accompanied by some model-specific properties. By inspecting a subset of the properties, namely the `numVertices` and `primCount` values, the hook detects when an enemy model is drawn and commences the wallhack. In this example, the values representing an enemy model are `24` and `12`.

The magic happens inside the `if()` statement. Using just a few lines of code, the wallhack draws the model in a way that ignores z-buffering, like so:

```
device->SetRenderState(D3DRS_ZENABLE, false); // disable z-buffering
DirectXHook::origDrawIndexedPrimitive(        // draw model
    device, primType, baseVertexIndex,
    minVertexIndex, numVertices, startIndex, primCount);
device->SetRenderState(D3DRS_ZENABLE, true);  // enable z-buffering
```

Simply put, this code disables z-buffering when drawing the enemy model and reenables it afterward. With z-buffering off, the enemy is drawn in front of everything.

### Changing an Enemy Texture

When a model is rendered onscreen, a *texture* is used to skin the model. Textures are 2D images that are stretched around 3D models to apply the colors and patterns that make up the model's 3D artwork. To change the way an enemy looks when it's drawn in your wallhack, you can set it to be drawn with a different texture, as in this example:

```
// when hook initializes
LPDIRECT3DTEXTURE9 red;
D3DXCreateTextureFromFile(device, "red.png", &red);
```

```
// just before drawing the primitive
device->SetTexture(0, red);
```

The first block of this code loads the texture from a file and is executed only once—when the hook is initialized. The full example code does this in an `initialize()` function, which gets called the first time the `EndScene()` hook callback is invoked. The second block of this code happens right before the call to the original `DrawIndexedPrimitive()` function in the wallhack, and it causes the model to be drawn with the custom texture.

### *Fingerprinting the Model You Want to Reveal*

The trickiest part to creating a good wallhack is finding the right values for `numVertices` and `primCount`. To do this, you can create a tool that logs every unique combination of the two variables and allows you to iterate over the list using your keyboard. Working example code for this tool won't be useful in the example application provided with this chapter, but I'll give you some high-level implementation details.

First, in the global scope, you'd declare a structure that has members to store the following:

- `numVertices` and `primCount`
- A `std::set` of this structure (let's call it `seenParams`)
- An instance of that structure (let's call it `currentParams`)

The `std::set` requires a comparator for this structure, so you'd also declare a comparison functor that calls `memcmp()` to compare two of the structures using `memcmp()`. Each time the `DrawIndexedPrimitive()` callback is invoked, your hack could create a structure instance with the intercepted values and pass it to a `seenParams.insert()` function, which should insert the parameter pair into the list only if the pair isn't already there.

Using the `GetAsyncKeyState()` Windows API function, you could then detect when the spacebar is pressed and execute something similar to this pseudocode:

```
auto current = seenParams.find(currentParam);
if (current == seenParams.end())
    current = seenParams.begin();
else
    current++;
currentParams = *current;
```

This would set `currentParams` to the next pair in `seenParams` when the spacebar is pressed. With this code in place, you could use code similar to a wallhack to change the texture of models matching `currentParams.numVertices` and `currentParams.primCount`. The tool could also draw those values on the screen so you could see them and write them down.

With a tool like this, finding the proper models is as easy as starting up a game in a mode where your character won't die (against a friend, in

a customization mode, and so on), running the bot, and pressing the spacebar until each model you need is highlighted. Once you have the values for your target models, you'll modify the `numVertices` and `primCount` check in your wallhack so it knows which models to highlight.

**NOTE** *Character models are commonly made up of smaller models for individual body segments, and games often show different models of a character at different distances. That means a game may have 20 or more models for one type of character. Even in that case, selecting only one model (say, the enemy's torso) to show in your wallhack may be enough.*

## Getting a Wider Field of Vision with Zoomhacks

Many games in the MOBA and real-time strategy (RTS) genres use a 3D top-down style that makes them immune to wallhacks. They also use darkness on the map as a type of fog, but showing the dark areas using a lighthack doesn't give any extra information; models hidden inside the fog are known only to the game server, not to the client.

This style makes most types of ESP hacks useless: there's little unknown information to reveal, so these hacks only augment your view of the information you can already see. One type of ESP hack, however, can still be helpful. *Zoomhacks* let you zoom out much farther than a game normally allows, effectively revealing large portions of the map that you couldn't see otherwise—and thus getting around the game's wallhack and lighthack immunity.

### Using NOPing Zoomhacks

MOBA and RTS games typically allow players a variable but limited amount of zoom. The simplest type of zoomhack finds the value of the *zoom factor* (a multiplier that changes as the zoom level changes, typically a `float` or `double`) and overwrites it with a larger value.

To find the zoom factor, fire up Cheat Engine and search for a `float` with an unknown initial value. (To brush up on Cheat Engine, head over to "Cheat Engine's Memory Scanner" on page 5.) For rescans, repeat the following process until there are only a few values left to find the zoom factor:

1. Go to the game window and zoom in.
2. Search for an increased value in Cheat Engine.
3. Go to the game window and zoom out.
4. Search for a decreased value in Cheat Engine.

Try to get the value list down to one option. To confirm that the remaining value is the zoom factor, freeze it in Cheat Engine and see how zoom behaves in-game; freezing the proper value will disable zooming. If you fail to find the zoom factor using a `float` search, retry the search using

a `double`. If both searches fail, try them again but correspond zooming in with decreased values and zooming out with increased values instead. Once you've found the zoom factor in memory, you can write a small bot to overwrite it to the zoom factor that best suits you.

More advanced zoomhacks NOP the game code responsible for making sure the zoom factor is within a set range. You should be able to find this code with OllyDbg. Set a memory on-write breakpoint on the zoom factor, zoom in-game to trigger the breakpoint, and inspect the code at the breakpoint. (To hone your OllyDbg memory breakpoint skills, flip to "Controlling OllyDbg Through the Command Line" on page 43.) You should see the code that modified the zoom factor. Zoom limitation code is typically easy to spot: constants that match the minimum and maximum zoom values are a dead giveaway.

If you can't find the limitation code using this method, then the limitation may be applied when the graphics are redrawn at a new zoom level, rather than when the zoom factor changes. In this case, switch your breakpoint to memory on-read and look for the same clues.

### Scratching the Surface of Hooking Zoomhacks

You can also create zoomhacks by using a Direct3D hook on the function `device->SetTransform(type, matrix)`, but this requires a deep understanding of how a game sets up the player's perspective. There are a few different ways to manage perspective, but you control zoom level using either *view* (transform type `D3DTS_VIEW`) or *projection* (transform type `D3DTS_PROJECTION`).

Properly manipulating transform matrices that control view and projection requires some pretty extensive knowledge of the mathematics behind 3D graphics, though, so I stay away from this method at all costs—and I've never had trouble simply manipulating the zoom factor. If you're interested in this kind of hack, though, I recommend reading a 3D game programming book to learn more about 3D mathematics first.

But sometimes, even a zoomhack isn't enough. Some useful information may remain hidden as a part of a game's internal state or may simply be hard for a player to determine at a moment's glance. For these situations, a heads-up display is the tool for the job.

## Displaying Hidden Data with HUDs

A *heads-up display (HUD)* is a type of ESP hack that displays critical game information in an overlay. HUDs often resemble a game's existing interface for displaying information like your remaining ammunition, a mini-map, your current health level, any active ability cooldowns, and so on. HUDs typically display either historical or aggregated information, and they're mostly used on MMORPGs. They are often text based, but some also contain sprites, shapes, and other small visual effects.

The HUDs you can create depend on what data is available in the game. Common data points are these:

- Experience gain per hour (exp/h)
- Creature kills per hour (KPH)
- Damage per second (DPS)
- Gold looted per hour (GPH)
- Healing per minute
- Estimated time until next level
- Amount of gold spent on supplies
- Overall gold value of items looted

More advanced custom HUDs may display large tables containing items looted, supplies used, the number of kills for each type of creature, and the names of players that have recently been seen.

Beyond what you've already learned about reading memory, hooking graphics engines, and displaying customized data, there's not much else I can teach you about how to create a HUD. Most games have a simple enough architecture that you can easily obtain most of the information you need from memory. Then, you can run some basic hourly, percentage, or summation calculations to get the data into a usable format.

### *Creating an Experience HUD*

Imagine you want a HUD that displays your current level, hourly experience, and how long you'll have to play before your character levels up. First, you could use Cheat Engine to find the variables that contain your level and experience. When you know those values, you can use either a game-specific algorithm or a hardcoded experience table to calculate the experience required to reach the next level.

When you know how much experience you need to level up, you can calculate your hourly experience. Put into pseudocode, that process might look like this:

```
// this example assumes the time is stored in milliseconds
// for seconds, remove the "1000 * "
timeUnitsPerHour = 1000 * 60 * 60
timePassed = (currentTime - startTime)
❶ timePassedToHourRatio = timeUnitsPerHour / timePassed
❷ expGained = (currentExp - startExp)
hourlyExp = expGained * timePassedToHourRatio

❸ remainingExp = nextExp - currentExp
❹ hoursToGo = remainingExp / hourlyExp
```

To find your hourly experience, `hourlyExp`, you'd store your experience and the time when your HUD first starts; these are `startExp` and `startTime`, respectively. This example also assumes `currentLevel` and `currentExp` are previously defined, where `currentLevel` is the character's level and `currentExp` is the current amount of experience.

With these values, `hourlyExp` can be calculated by multiplying a ratio ❶ of the time units in an hour to the time that has passed by the experience gained since `startTime` ❷. In this case, the time unit is a millisecond, so the time units get multiplied by 1,000.

Next, `currentExp` is subtracted from `nextExp` to determine the remaining experience ❸ to level up. To calculate how many hours are left to level up, your remaining experience is divided by your hourly experience ❹.

When you have all this information, you can finally display it onscreen. Using the Direct3D hooking engine provided in this book's example code, you'd draw the text using this call inside the `EndScene()` hook callback:

```
hook->drawText(
    10, 10,
    D3DCOLOR_ARGB(255, 255, 0, 0),
    "Will reach level %d in %0.20f hours (%d exp per hour)",
    currentLevel, hoursToGo, hourlyExp);
```

That's all you need for a working, experience-tracking HUD. Variations of these same equations can be used to calculate KPH, DPS, GPH, and other useful time-based measures. Furthermore, you can use the `drawText()` function of the Direct3D hook to display any information you can locate and normalize. The hook also contains `addSpriteImage()` and `drawSpriteImage()` functions that you can use to draw your own custom images, allowing you to make your HUDs as fancy as you want.

### *Using Hooks to Locate Data*

Memory reading isn't the only way to get data for a custom HUD. You can also gather information by counting the number of times a specific model is drawn by the `DrawIndexedPrimitive()` function, hooking the game's internal functions responsible for drawing certain types of text, or even intercepting function calls responsible for processing data packets from the game server. The methods you use to do this will be drastically different for every game, and finding those methods will require you to pair everything you've learned from this book with your own ingenuity and programming instincts.

For instance, to create a HUD that displays how many enemies are on the map, you could use the model-fingerprinting methods used by wallhacks to count the number of enemies and output that number to the screen. This method is better than creating a way to read the list of enemies from memory, since it doesn't require new memory addresses every time the game patches.

Another example is displaying a list of enemy cooldowns, which would require you to intercept incoming packets that tell the client which spell effects to display. You could then correlate certain spells with certain enemies based on spell and enemy location, spell type, and so on, and use that information to track spells each enemy has used. If you correlate the data with a database of cooldown times, you can display exactly when each enemy spell can be used again. This is especially powerful because most games don't store enemy cooldowns in memory.

## An Overview of Other ESP Hacks

In addition to the hacks discussed in this chapter, there are a number of ESP hacks that don't have common names and are specific to certain genres or even certain games. I'll quickly take you through the theory, background, and architecture of some of these hacks.

**Range Hacks**

Range hacks use a method similar to wallhacks to detect when the models for different types of champions or heroes are drawn. Then they draw circles on the ground around each hero model. The radius of each circle corresponds to the maximum attack range of the champion or hero it surrounds, effectively showing you areas where you can be damaged by each enemy.

**Loading-Screen HUDs**

Loading-screen HUDs are common in MOBA and RTS games that require all players to sit through a loading screen while everyone's game is starting up. These hacks take advantage of the fact that such games often have websites where historical player statistics can be queried. You can write a bot that automatically queries the statistics of each player in the game and seamlessly displays the information as an overlay on your loading screen, allowing you to study your enemies before launching into battle.

**Pick-Phase HUDs**

Pick-phase HUDs are similar to their loading-screen cousins, but they are displayed during the pregame phase when each player is picking a champion or hero to play. Instead of showing enemy statistics, pick-phase HUDs show statistics about allies. This allows you to quickly assess the strengths and weaknesses of your allies so you can make better decisions about which character to play.

**Floor Spy Hacks**

Floor spy hacks are common in older 2D top-down games that have different distinct floors or platforms. If you're on the top floor, you might want to know what's going on downstairs before you go charging in. You can write floor spy hacks that modify the current floor value (typically an `unsigned int`) to a different floor above or below you, allowing you to spy on other floors.

Games often recalculate the current floor value every frame based on player position, so NOPs are sometimes required to keep the value from being reset every time a frame is redrawn. Finding the current floor value and the code to NOP would be similar to finding the zoom factor, as discussed in "Using NOPing Zoomhacks" on page 197.

## Closing Thoughts

ESP hacks are powerful ways to obtain extra information about a game. Some of them can be done pretty easily through Direct3D hooks or simple memory editing. Others require you to learn about a game's internal data structures and hook proprietary functions, giving you a reason to employ your reverse engineering skills.

If you want to experiment with ESP hacks, study and tweak the example code for this chapter. For practice with more specific ESP hacks, I encourage you to go out and find some games to play around with.