Automating *Super Hexagon* Gameplay

ecx86[1]

***Abstract*—*Super Hexagon* is a 2012 video game developed by Terry Cavanagh, in which players use one-dimensional directional inputs to avoid incoming obstacles. In this paper, we present a method to automate playing the game, using DLL injection to directly interface with the game.**

I. Introduction

Super Hexagon is a simple game where the game field is laid out using a polar coordinate system [1]. The left and right arrow are used to adjust the player's argument (angle); however, the player's magnitude (distance from center) is fixed. Obstacles have a fixed argument and steadily decreasing magnitude. It is the player's responsibility, then, to constantly adjust his or her argument to navigate the incoming obstacles.

Fig. 1: A screenshot of typical *Super Hexagon* gameplay. The player entity, the small triangle in the center, tries to avoid the incoming obstacles. Global rotation is disabled, and each sector is labeled with its internal index.

The game adds further difficulty by constantly rotating the entire game field at random speeds. With global rotation disabled, however, one notices that obstacles are placed among six equal sectors. Our job, then, to choose which sector to move towards in order to avoid hitting an obstacle and subsequently feed the corresponding input to the game.

II. Interfacing with the Game

To discern the player's current sector (Figure 1), as well as the sector and distance of the obstacles, we opt to use a code injection approach, as opposed to a graphical or visual approach. This sidesteps the graphical noise thrown in by the game, such as global rotation, scaling, tilting, and flashing. It also allows us to directly feed in keyboard inputs via the GLUT keystate vector [2]. Finally, using code injection, we can hook internal game functions to properly synchronize our program's update loop with that of the game.

To do this, we implement the ubiquitous LoadLibraryA DLL injection [3]. A brief summary of the process is to:

1) OpenProcess to open a handle to the game process.
2) VirtualAllocEx to allocate a small buffer to hold the path to the DLL in the game's address space.
3) WriteProcessMemory to fill the buffer.
4) CreateRemoteThread to call LoadLibraryA and the DLL path buffer as the argument to load our DLL.

At this point, our DLL is now executing in the game's address space, and consequently can directly read and write memory. In the DLL's main function, we spawn a new thread to avoid blocking the main thread and proceed.

A. Locating the Game State Information

Before accessing or mutating the game state, we must first locate it in memory. Through reverse engineering, we notice that the bulk of the game's state information is contained in the *Superhex* class. If we could consistently locate a pointer to this class, we can rely on its structure offsets to access the game state through its fields. Luckily, we find a short assembly snippet in `main` which saves a pointer to the Superhex instance in a memory location `pSuperhex`:

```
.text:00439EAC call    InitSuperHex
.text:00439EB1 jmp     short .text:00439EB5
.text:00439EB3 xor     eax, eax
.text:00439EB5 push    eax
.text:00439EB6 mov     [ebp+var_4], 0FFFFFFFFh
.text:00439EBD mov     pSuperHex, eax
```

We simply search the `.text` segment in memory for this code. Then, we do an offset double dereference to acquire the Superhex pointer:

```
HMODULE hSuperhex = GetModuleHandleA(NULL);
DWORD dwSigLoc = memmem((DWORD) hSuperhex + 0x20000, 0x30000,
↪   "\xEB\x02\x33\xC0\x50\xC7\x45", 7);
CSuperhex *pSuperhex = **(CSuperhex***)(dwSigLoc + 0xD);
```

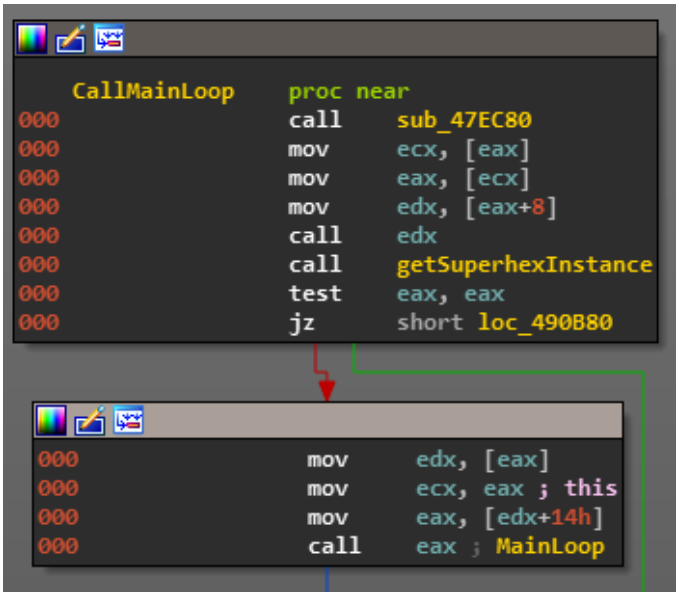
Now we are able to modify the game state, call game functions, and hook class methods.

B. Hooking the Update Function

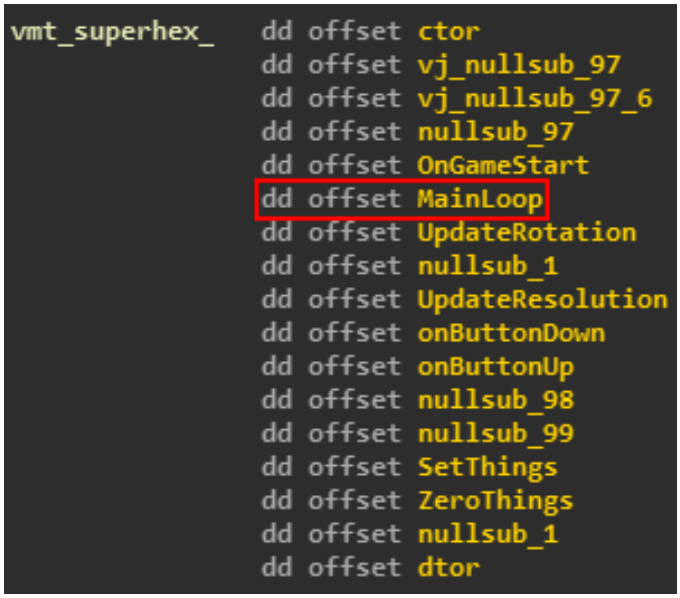
Reverse-engineering the game, we notice that the game's main update function *MainLoop* is called via a virtual method table (VMT), i.e. MainLoop is a virtual method of theSuperhex class.

This makes hooking the update function easy; we simply implement a classic VMT hook [4]. Two approaches are

[1]ecx86@users.noreply.github.com



(a) The *Superhex* instance pointer is loaded into eax, then the 6th virtual method (MainLoop) is called.



(b) An excerpt of *Superhex* class's virtual method table.

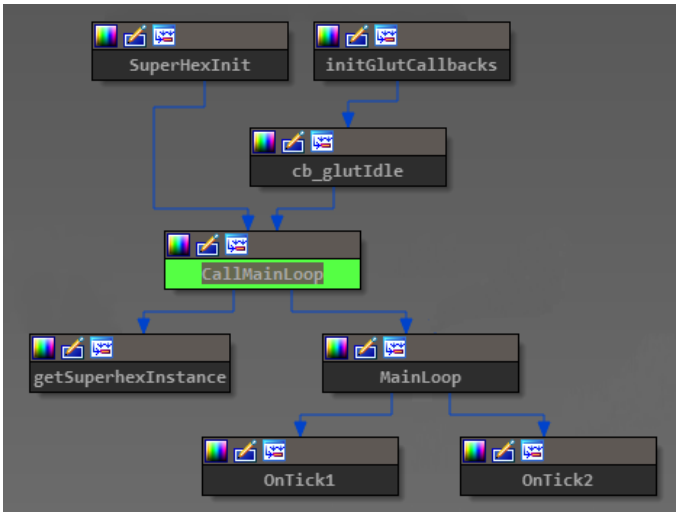


Fig. 2: Callgraph of cross-references to MainLoop.

possible: we can either replace MainLoop's function pointer in the VMT, or we can swap out the entire VMT by replacing the VMT pointer. We opt for the second approach, as it makes unhooking easier because only the VMT pointer must be restored. More technically, we first clone the VMT, patch the function pointer at the relevant index, and finally replace the VMT pointer with one to our hooked copy. In code:

```
// Store pointer to class, which is also the pointer to its VMT
m_pClassVMT = (PDWORD*) pClass;
// Store the pointer to the original VMT
m_vmtOld = *m_pClassVMT;
// Copy old vmt to our new copy
m_vmtNew = calloc(m_nFuncs, sizeof(DWORD));
memcpy(m_vmtNew, m_vmtOld, sizeof(DWORD) * m_nFuncs);
// Hook 6th vfunc
m_vmtNew[0x14 / sizeof(DWORD)] = (DWORD) cbMainLoop;
// Swap VMT pointers
*m_pClassVMT = m_vmtNew;
```

Our hook callback will execute as soon as MainLoop is called; consequently, we must remember to call the original function, but not through the VMT, as that would lead to infinite recursion. Under the __thiscall calling convention, the class instance pointer is passed in the ecx register. We emulate that using __fastcall in our callback.

```
typedef int(__thiscall *MainLoop)(CSuperhex *pThis);
MainLoop pMainLoop = m_vmtOld[0x14 / sizeof(DWORD)];
int __fastcall cbMainLoop(CSuperhex *pThis) {
        AiOnTick(pThis); // call our bot to make a move
        return pMainLoop(pThis); // call original function
}
```

Finally, we are now equipped to write the functional part of the program proper.

III. Gameplay Decision-Making

Previously, we stated that the game's rotational argument space is partitioned into 6 equal segments. This is not strictly true, as occasionally the game will reduce the number of segments to 4 or 5 (Figure 3). This is not a big deal, as we can easily determine the number of axes through a struct member `pSuperhex->gamestate.axisCount` at offset 0x1BC.

Each tick, we compute some facts about the magnitude-wise geometry of each section (1). Then, we assign scores to each section, and move towards the best section (2). If our current section is the best section, we align ourselves to the center of the section, anticipating a future move left or right (3). Finally, we feed input to the game (4).

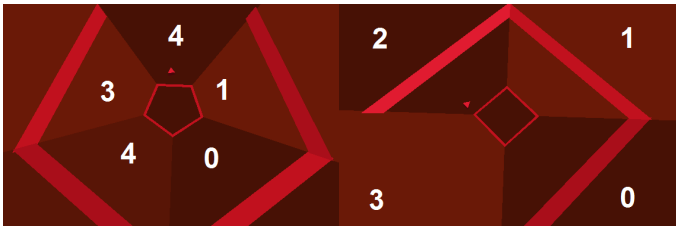


Fig. 3: The 5-axis and 4-axis cases.

```
int ceilings[6];
int floors[6];
int sides = pSuperhex->gamestate.axisCount;
float playerSection = (pSuperhex->gamestate.playerRotation) *
↪  sides / 360.f;
int maxDist = 145 - pSuperhex->gamestate.wallSpeed; // distance
↪  below which walls no longer hurt us
int hedge = maxDist * (360 / sides / 10) + 50; // min space
↪  required to fit through section

for (int i = 0; i < sides; i++) floors[i] = maxDist;
for (int i = 0; i < sides; i++) ceilings[i] = 999999;
for (int i = 0; i < pSuperhex->gamestate.wallCnt; i++) {
        wall_t wall = pSuperhex->gamestate.walls[i];
        if (!wall.enabled) continue;
        int section = wall.section;

        int startDist = wall.distance;
        int endDist = wall.distance + wall.width;
        if (startDist <= floors[section] && endDist >
        ↪  floors[section]) // is it blocking our path
                floors[section] = endDist;
        if (startDist < ceilings[section] && wall.distance > 0
        ↪  && startDist > maxDist) // how narrow it is
                ceilings[section] = startDist;
}
```

Listing 1: Computing section geometries.

```
int evaluateSection(int candPos, int curPos, int dir) {
        int startCeiling = ceilings[curPos];
        int penalty = 0;
        for (; curPos != candPos; curPos = (curPos + dir +
        ↪  sides) % sides) {
                if (floors[(curPos + dir + sides) % sides] >
                ↪  ceilings[curPos]) { // it's blocked
                        penalty += 6 * (floors[(curPos + dir +
                        ↪  sides) % sides] -
                        ↪  ceilings[curPos]);
                }
                if (floors[curPos] + hedge > ceilings[(curPos +
                ↪  dir + sides) % sides]) { // too tight
                        penalty += 4 * (floors[curPos] + hedge
                        ↪  - ceilings[(curPos + dir + sides)
                        ↪  % sides]);
                }
                penalty += 30;
        }
        return ceilings[candPos] - startCeiling - penalty;
}

int bestPos, bestScore, bestDir = (int) playerSection, 0, 0;
for (int i = 0; i < sides; i++) {
        for (int dir = -1; dir < 2; dir++) {
                if (dir == 0) continue;
                int score = evaluateSection(i, playerSection,
                ↪  dir);
                if (score > bestScore) {
                        bestScore = score;
                        bestDir = dir;
                        bestPos = i;
                }
        }
}
```

Listing 2: Assigning scores to sections.

```
if (bestDir == 0) {
        float pos = fmodf(playerSection, 1.f) - .5f;
        if (pos > .1f) bestDir = -1;
        else if (pos) bestDir = 1;
}
```

Listing 3: Align to middle of section.

```
pSuperhex->buttonStates[LeftArrow] = bestDir == 1;
pSuperhex->buttonStates[RightArrow] = bestDir == -1;
```

Listing 4: Feed input by writing to GLUT keystate vector.

And with that, the bot's functionality is complete.

IV. Conclusion

We implemented an internal algorithm to automate playing the video game *Super Hexagon*. To inject code into the game's address space, a typical LoadLibraryA DLL injection was used. A signature scan, coupled with reverse-engineered structure offsets, allowed us to effectively interface with the game once inside its address space. Next, a virtual method hook was used to execute the bot's algorithm each time the game updated. A primitive scoring heuristic was used to assess future moves and decide whether to move clockwise, counterclockwise, or neither. Regrettably, the parameters for the fitness function were hand-picked, and a future extension could be to use machine-learning constructions such as gradient descent and neural networks to automatically generate them.

Acknowledgment

Firstly, thank you for taking the time to read this technical report. I know it's dense, so I appreciate you for sticking through it. Secondly, I would like to thank Terry Cavanagh for making such a great game. I had as much fun playing *Super Hexagon* as I did making this project.

References

[1] http://store.steampowered.com/app/221640/Super_Hexagon/

[2] https://github.com/google/liquidfun/blob/master/freeglut/include/GL/freeglut_std.h

[3] http://www.coderbag.com/Threading/DLL-Injection-Using-Remote-Thread

[4] https://en.wikipedia.org/wiki/Hooking#Virtual_Method_Table_hooking