Our first submission here, we hope you will all enjoy it as much as we did while making it!

This is a double submission. We were unsure whether to include the Training levels or not, so we will let the community decide. This is the version WITHOUT the Training levels, the submission with them is available [\_\_TODO: link to the other submission\_\_|here].

----

[*https://www.youtube.com/watch?v=w4gxOXmBLd8*](https://www.youtube.com/watch?v=w4gxOXmBLd8)

%%TOC%%

Released in 2001 as one of the launch titles of the GBA, ''*Kuru Kuru Kururin*'' is an unusual game that is a mix of racing/puzzle genres. The player controls a slowly spinning stick called the Helirin, and must maneuver it through a series of mazes. The player controls the direction and speed of movement but not the Helirin's constant rotation.

This TAS is a significant improvement to the [1307M|2009 movie by Ryuto] due to game-changing discoveries: wall clipping and out-of-bounds goal zones.

**!!!Categories**

\*Emulator used: Bizhawk 2.3.1, mGBA core

\*Aims for fastest time

\*Takes damage to save time

\*Heavy glitch abuse

\*Genre: Puzzle

\*Genre: Racing

**!!!Rules**

\*Reach the credits as fast as possible

\*All glitches allowed

This means that we have to complete the thirty “main” levels of the game while saving the bird trapped in the third level of each world.

**!!!The basics**

**!!Objective of a level**

Reach the goal zone (=yellow zone).

When you hit a wall, you loose a heart. If you loose all your three hearts, the Helirin breaks and you must restart the level from the beginning. In some levels there are heart zones on your way that heal you.

Sometimes there is a collectible item in the level, which is either a cosmetic bonus or a bird. Only birds are necessary to reach the credits.

**!!Position**

Position is given by a triple (X,Y,R) with X,Y the coordinates and R the angle of the Helirin.

\*X and Y are 32-bit values. Both can be interpreted as a triple (pixel,subpixel,subsubpixel) with the pixel in the range [[-32768..32767]] and both subpixel and subsubpixel in the range [[0..255]]. Subsubpixels are rarely relevant, so we will mostly be working with the couple (pixel,subpixel).

\*R is a 16-bit value in the range [[0..65535]]. 0 and 32768 are vertical.

Axes:

― — > X

|

|

v

Y

**!!Speed under ”normal” behavior**

X/Y speed depends on how many A/B buttons are held. The more buttons you hold, the faster you go.

The table below gives examples of a horizontal input and a diagonal input. Speed values are in (pixel,subpixel) format.

||Direction|| #A/B held || X speed || Y speed ||

| Right | 0 | +1,128 | 0 |

| Right | 1 | +2,64 | 0 |

| Right | 2 | +3,0 | 0 |

| Down-right| 0 | +1,15.5 | +1,15.5 |

| Down-right| 1 |+1,151.25 |+1,151.25 |

| Down-right| 2 | +2,31 | +2,31 |

Holding buttons makes you respectively 1.5x and 2x faster.

Rotation is at a constant +/-182 rate (+ is clockwise).

**!!Collision**

When you hit a wall, you get pushed back. It has an impact on all three parameters:

\*you get an X/Y bump speed that decreases with time (it takes them about 30 to 40 frames to get back to zero).

\*your rotation speed is set to +/-1024 and gradually gets back to its initial value (it takes about 15 frames). When collision lasts two or more consecutive frames, rotation speed switches back and forth between +1024 and -1024, which could result in your rotation being sped up instead of being pushed back.

**!!Springs**

Some levels include blue springs. They are the only way to switch between clockwise and counterclockwise motion. They can also be used to speed up rotation.

**!!Moving objects**

Some levels include cylinders and spiked balls. They hurt. Their movement is cyclic and fixed: if you hit them, they will not deflect.

**!!Horns**

The best feature of the game. While in a level, press L or R to have fun. Holding A/B while pressing L or R gives other horn sound effects for a grand total of six.

## !!!Glitches and tricks used

**!!Invulnerability frames**

Whenever you take damage, the game gives you 20 frames of invulnerabilty. This allows to hit a wall multiple times and only loose one heart, which is especially useful to set up rotation and to perform wall “surfing” (detailed below).

\_\_TODO: gif?\_\_

**!!Wall clipping**

When colling with a wall, the game does not behave in the same way depending on which part of the Helirin is colliding. In particular if it is the center, then the game will only rely on your last input and push you in the oppposite direction. But simply going towards the wall would not work, as the game would push you in the opposite direction. So, in order to get the center into the wall, we typically use the bump speed from an earlier collision. This allows then to not make any input while still moving towards the wall. A tiny flaw in the game physics with huge consequences!

Also a proof that walls are inefficient to prevent border crossings.

\_\_TODO: slow-mo gif\_\_

**!!Wall surfing**

Once the center of the Helirin is in a wall, as the game only takes into account yout last input and push you the other way, pressing inputs parallel to the wall makes you travel along the wall at very high speed.

\_\_TODO: gif\_\_

**!!Out-of-bounds stuff**

Because of the way collisions are computed (see section “Technical details”), some objects are ''physically'' replicated: even though they are not visible, you can interact with them.

\*Replicated: walls and starting/heart/goal zones

\*Not replicated: birds, moving objects, springs, and cosmetic bonuses

Replicated goal zones are especially interesting. Some of them are closer to the starting zone than the regular goal zone. This has been a major dicovery since they have been used in about half of the levels. See section “Times per level” for a recap of where they have been used.

**!!(No) restart trick**

The [1307M|2009 movie by Ryuto] restarted the game after completing the third level of each world. These levels induced an additional cutscene because of saving a bird, which was skipped by restarting the game (saving 68 frames each time). Doing this after any other level would loose 24 frames instead.

However, it looked like this was only made possible because of using emulator VBA-rr 19.3, which skipped the Game Boy introduction. As Bizhawk 2.3.1 does not skip it by default and soft resetting is not a thing in this game, we did not include this trick in our movie.

## !!!Tools

**!!Memory watch**

A .wch file is available [https://github.com/E-Sh4rk/KururinTAS|here].

|| Name || Address (IWRAM)|| Size || Description ||

| X / Y | 0x4544 / 0x4548 | 32 bits | Position of the center of the helirin. If we only look at the 16 most significant bits, we get the position in pixels. |

| XB / YB | 0x454C / 0x4550 | 32 bits | Bump speed. A bump speed is applied when the helirin hits a wall, then it decreases gradually. |

| XS / YS | 0x4554 / 0x4558 | 32 bits | Input speed. It only depends on the direction pressed at the previous frame. |

| Angle | 0x4572 | 16 bits | Angle of the helirin, 0 and 32768 being vertical. 2⸢⸢16⸣⸣ corresponds to 360°. For instance: (90 / 360) \* 2⸢⸢16⸣⸣ = 16384 is 90°. |

| Angle Rate | 0x4574 | 16 bits | Rotation speed. It is 182 or -182 by default, but it momentarily changes when the helirin hits something. |

| Default Rate | 0x4576 | 16 bits | 182 when the helirin rotates clockwise, and -182 when it rotates counter-clockwise. Can change when hitting a spring. |

| Invulnerability | 0x4585 | 8 bits | Number of invulnerability frames left. Grows to 20 when the Helirin looses a heart, then decreases by 1 every frame until 0. This value is decremented before being used, so having it to 1 is equivalent to having it to 0. |

| MapW and MapH | 0x313C / 0x313E | 16 bits | Size of the map in a number of tiles. A tile is 8x8 so multiplying these numbers by 8 gives the size of the map in pixels. |

| Collision Mask | 0x45D4 | 32 bits | Indicates which parts of the helirin are in collision with a wall. |

**!!TAStudio**

As this game does not have any known desync or lag issue, progress was nonlinear and we just had to glue the segments together at the end. This has made TAStudio a very efficient and comfortable interface to work with.

**!!Maps**

The maps of the thirty “main” Adventure Mode levels (from Grasslands 1 to Ghost Castle 3) are available on [https://www.vgmaps.com/Atlas/GBA/index.htm#K|vgmaps].

**!!Speed HUD**

Lua script written by ThunderAxe31. Displays X/Y position, speed and bump speed. Available [=userfiles/info/49530161300647551|here].

\_\_TODO: add image\_\_

**!!OOB viewer**

Lua script created by ThunderAxe31, then improved in conjoint work with E-Sh4rk. Available on [https://github.com/E-Sh4rk/KururinTAS/tree/master/OoB%20Viewer|Github].

\_\_TODO: add image\_\_

**!!KuruBot**

Bot designed by E-Sh4rk. Efficient and customizable tool to help finding new strategies in Kuru Kuru Kururin’s levels (more details in the “Technical details” section). Available on [<https://github.com/E-Sh4rk/KururinTAS/tree/master/KuruBot>|Github].

Example video [https://youtu.be/GmkBdvF6ZxM|here].

## !!!Technical details

**!!Game physics**

Before we give an overview of what the game performs every frame, we need to develop a couple notions.

----

''Collision mask''

Even though the Helirin is 63 pixels long, only 17 points are checked for collision. These points, which we will call ''collision points'', are marked as “O” below. (O) is the center, which is also a collision point.

O⁃⁃O⁃⁃⁃O⁃⁃⁃O⁃⁃⁃O⁃⁃⁃O⁃⁃⁃O⁃⁃⁃O⁃⁃⁃(O)⁃⁃⁃O⁃⁃⁃O⁃⁃⁃O⁃⁃⁃O⁃⁃⁃O⁃⁃⁃O⁃⁃⁃O⁃⁃O

In order to keep track of which points are in collision, the game uses a ''collision mask''. The collision mask is a 32-bit value that must be read in binary: each bit tells if a certain point is in collision with something. If we number collision points in the following way:

16⁃⁃14⁃⁃⁃12⁃⁃⁃10⁃⁃⁃8⁃⁃⁃6⁃⁃⁃4⁃⁃⁃2⁃⁃⁃(0)⁃⁃⁃1⁃⁃⁃3⁃⁃⁃5⁃⁃⁃7⁃⁃⁃9⁃⁃⁃11⁃⁃⁃13⁃⁃15

then (with 0 being the least significant bit): bit n°i is 1 when point i is in collision with something.

(Note that the bits 17 to 31 are not used.)

----

''Collision detection & Map''

To detect if a point is in collision with a wall, the game uses a map of 8x8 tiles.

The map is stored at the beginning of the EWRAM (0x02000000). The two first 16-bit words contain the width and height of the map, and then (at address 0x02000004) there is a matrix describing the map (as a sequence of 16-bit tile identifiers).

Thus when the game must determine the tile behind a certain point, it reads the 16-bit identifier at the following address:

0x02000004 + 2\*(y\_tile % map\_height)\*map\_width + 2\*(x\_tile % map\_width)

with:

\*''map\_width'' and ''map\_height'' the dimensions of the map. They are respectively available at address `0x02000000` and `0x02000002`.

\*''x\_tile'' the x-coordinate of the point in terms of number of tiles. A tile being 8x8 pixels, it is computed this way: ''x\_tile'' = (''x'' >> 16)/8, with ''x'' the 32-bit X-coordinate of the point (the ">> 16" operation is to discard the subpixel and subsubpixel parts and only keep the pixel part).

\*\*Similarly: ''y\_tile'' = (''y'' >> 16)/8.

Note that the game only keeps track of the coordinates of the center. To compute the coordinates of the other collision points, the game uses the angle value R and a precomputed table of cosinus/sinus values.

----

''Speed''

Speed can be decomposed in the following way:

\*''Rotation speed'':

\*\*constant +/-182 under normal conditions, temporarily changed to (+/-)1024 when hitting a wall, then returns to its initial value by steps of 91 per frame.

\*X/Y speed:

\*\* ''I*nput speed:*''based on the input. The norm is 3, 2.25 or 1.5px/frame depending on the number of A/B buttons that are held. It changes instantly (no inertia, instant acceleration). If applying this speed would make the Helirin hit a wall, then it is canceled (so the player has no control when the Helirin hits a wall).

\*\* ''B*ump speed:*'' applied when the Helirin hits a wall. The norm is 2px/frame. It takes the same direction as the vector starting from the collision point to the center of the helirin. It is then multiplied by 3/4 each frame until reaching 0. If both sides of the helirin are in collision at the same time, then no bump speed is applied (which sounds reasonable since we cannot determine in which direction to apply it).

\*\* ''W*all speed:*'' only applied when the center of the Helirin hits a wall. The norm is 4px/frame. It is applied in the opposite direction of the current input. It shares the same two variables as the bump speed. It was probably introduced to avoid wall clips, but it actually had the opposite effect, allowing the ''wall surfing'' trick.

The theoretical max X/Y speed is 9px/frame. However in practice bump speed and wall speed are hardly ever triggered at the same time and in the same direction, so the highest speed value that can be obtained consistently is actually 7px/frame (with wall surfing).

----

As we promised, here is now an overview of what is performed at each frame:

#If not zero, then the bump speed variable is multiplied by 3/4. If different from the current default rotation speed (+ or -182), then the rotation speed gets closer to it by 91. If not zero, then the invulnerability variable is decreased by 1.

#The position of the helirin is updated by applying input speed, bump speed and rotation speed.

#If the center of the Helirin is in a goal zone, then the level is completed. If it is in a healing zone, then all hearts are restored (and the Helirin is marked safe for the rest of the frame).

#The collision mask is computed. The collisions with the springs, birds and bonuses are also computed but they will not be detailled here.

#If there is collision with a wall/moving object and if the invulnerability variable is zero, then the Helirin looses a heart and the invulnerability variable is set to 20. If there is no heart left, then the level is failed. This step is not performed if the helirin is in a healing zone.

#If there is a collision with a wall (in other words, if the collision mask is not zero), then the input speed that was applied at step 2 is reverted and the rotation speed is set to +1024 (if it was negative before) or -1024 (if it was positive before). If only one side of the Helirin is in collision, then the corresponding bump speed is computed and assigned to the bump speed variables, and this bump speed is immediatly applied to the Helirin.

#If at least one of the physical points 0, 1 or 2 is in collision with a wall, and if a direction is pressed, then corresponding wall speed is computed and assigned to the bump speed variables, and this wall speed is immediatly applied to the Helirin.

For a more precise description, you can refer to the [https://github.com/E-Sh4rk/KururinTAS/blob/master/KuruBot/KuruBot/KuruBot/Physics.cs|implementation of the KuruBot].

Note that we have \_\_not\_\_ considered moving objects (we have had no issue with them, so we were lazy :p). But as long as moving objects are not in the way, we have encountered no counterexample yet.

**!!OOB stuff**

Recall how collisions were checked in memory (subsection ''Collision detection & Map''):

0x02000004 + 2\*(y\_tile % map\_height)\*map\_width + 2\*(x\_tile % map\_width)

Collisions were computed modulo the dimensions of the map, which means that some objects are ''physically'' replicated: even though they are not visible, you can interact with them. In particular:

\*Replicated: walls and starting/heart/goal zones

\*Not replicated: birds, moving objects, springs, and cosmetic bonuses

However, the way walls and zones are replicated is very different as you can see in these examples:

\_\_TODO: add Grasslands 1 and Machine Land 1 image examples\_\_

Here is an explanation:

\*When computing collisions with ''walls'', the game considers the X/Y coordinates of the Helirin to be ''unsigned''. It results in an overflow at the left and at the top of the map. When the width of the map is not a power of 2, the x-coordinate overflow will change the result of the modulo. This explains the shift of the walls in Machine Land 1.

\*When computing collisions with ''starting/healing/ending zones'', the game considers the X/Y coordinates of the Helirin to be ''signed''. Thus there is no overflow at the left nor at the top \_\_BUT\_\_ as “y\_tile % map\_height” is negative in the top OOB duplicate (yeah, the C modulus is weird), this gives an address outside of the matrix storing the tiles. This explains why there is no starting/healing/ending zone in the top OOB duplicate. The slight vertical shift of these zones in the left OOB duplicate can also be explained this way. (This slight shift is especially interesting as it allows to reach an OOB goal zone without loosing an additional heart. See example [\_\_TODO: link\_\_ |in Star Land 3]).

**!!KuruBot**

Here is a legitimate question: why would anyone bother to recode the game physics when there is an emulator available? Well, simulating through Bizhawk would not be very efficient, as search speed is bounded by emulator speed. But we do not need most of what the emulator is computing (sound, graphics), all we need to know is the update on speed and position. Thus re-implementing the physics allows to keep computations at their minimum and thus explore dozens of thousands of states per second. This is KuruBot’s reason for existence.

But is it enough? Let’s get a rough estimate of the size of the search space:

\*X and Y are 32-bit values but not every pixel will actually be visited, so let’s say 24 bits each.

\*Both “bump speed” 32-bit variables (one for X, one for Y) are key to get wall clips so they have to be considered. Let's say about half of the bits would actually be relevant (16 bits each).

\*Any angle could be useful a priori, so 16 bits.

\*Rotation speed ranges between -1024 and +1024 by steps of 91, so it takes about 2048 / 91 ≈ 23 different values, say 4 bits as a lower bound.

\*The couple (#hearts, invulnerability count) can take 3\*20=60 different values, but let’s say only about half of them would be relevant (5 bits).

This would result in about 2^105 ≈ 10^31 different states! Thus unfortunately there is \_\_no way\_\_ to get \_\_exact\_\_ solutions, even by keeping computations to their minimum.

But this is not the end of the journey. This just means that we have to renounce to reach perfection and instead try to find clever ways to reduce state space with minimal impact on optimality. To do so, we have had two main ideas:

\*reduce precision on every variable (X/Y position, rotation, speed). For instance, X is a 32-bit value but if two states only differ by less than 1/64 px, then there is very little chance that this would make a difference (even for wall clips, where bump speed is typically about 1/10 px/frame). Thus a state could be ignored by the bot if an ''*almost similar*'' state has already been visited. We could also adapt the precision according to the situation. For instance we could set a small precision by default and gradually increase precision around walls, as wall clips need precision.

\*use a custom A\* as our shortest path finding algorithm. A\* relies on a cost map to guide the search. So if we have a good guess on what an optimized path should look like, then we can prioritize the search on the corresponding area. For instance if we wanted to perform a tricky wall clip, we could set a high value on walls in order to prioritize the search on them.

KuruBot implements all these ideas. It has been written in C# and the game physics described above has been implemented. The parameters for the cost map of A\* and the reduction rates are gathered in a configuration file. Some examples of configuration files are given in the[<https://github.com/E-Sh4rk/KururinTAS/releases>|releases].

KuruBot being operational, our work was not over. We as TASers still had to tune the parameters, find the right balance between precision and time complexity, decide for a global path, break it into small segments and only use KuruBot in these smaller segments. Here is a typical situation when using KuruBot:

- ''**Me:**'' Please KuruBot, try to go from this position to the ending zone.

- ''**KuruBot:**'' *\*10 GB of RAM and 5min later\* ...*

- ''**Me:**'' KuruBot, abort. Try to reach this little area instead. And please keep 2 hearts! It will be useful later.

- ''**KuruBot:**'' *\*3 sec later.\** Done.

- ''**Me:**'' Now, please go from this new position to the ending zone by clipping through the wall.

- ''**KuruBot:**'' *\*3 sec later.\** Done.

- ''**Me:**'' Thank you, KuruBot. You are so smart. I love you. I mean, really. We can do so much together!

- ''**KuruBot:**'' *...*

What has worked best from our experience:

\*low default precision with increased precision around walls

\*heuristic flooding-based cost map with additional reward on walls

\*have at most one wall clip per segment

Note that KuruBot could also be used for other purposes than fastest completion, like damageless levels.

**!!(obsolete) Pixel and subpixel adjustment**

Before the “KuruBot era”, getting wall clips required precise setups, as bump speeds were rarely above a few dozen subpixels when getting close to walls. As input speed is always greater than 1px/frame whatever inputs you do, it can be difficult to reach a precise range of positions. But it turns out that with help from diagonal inputs, it is possible to adjust precisely the Helirin position within a few frames.

Below are some examples. They were heavily used in the [\_\_TODO: link\_\_|v0 movie]. You can recognize some of them if you take a closer look at the inputs done right before some of the wall clips. Who knows, they might become useful again someday!

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Notation:

-- X(pixel,subpixel)

-- "R2": R = direction in {U,L,D,R,UL,UR,DL,DR}

2 = speed in {0,1,2} (=number of held buttons in {A,B})

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Methods to change X pixel with minor impact on X subpixel:

DL2+UL2 [[-4, 62]]

R1 [[+2, 64]]

R2 [[+3, 0]]

=> [[+1, 2]] using 4 frames

Note: with reverse moves: [[-1, 2]] using 4 frames

Note: there is a similar method for Y

\*

DR2+UR2 [[+4, 62]]

L1 [[-2, 64]]

=> [[+1,254]] using 3 frames

Note: it can be reversed + similar method for Y

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Methods to adjust X subpixel:

DL0+UL0 [[-2,31]]

R1 [[+2,64]]

=> [+0, 33] using 3 frames

\*

DR1+UR1 [[+3,46]]

L2 [[-3, 0]]

=> [+0, 46] using 3 frames

\*

DL2+UL2 [[-4, 62]]

2\*R1 [[+4,128]]

=> [+0, 66] using 4 frames

\*

DL2+UL2 |[-4, 62]]

DR1+UR1 [[+3, 46]]

R0 [[+1,128]]

=> [+0,112] using 5 frames

\*

DR0+UR0 [[+2, 31]]

L0 [[-1,128]]

=> [[+0,159]] using 3 frames

Note: they can be reversed + similar methods for Y

## !!!Times per level

**!! Why not IGT?**

Although the game gives an in-game time per level, we decided to go for real-time. The main reason is that the game gives a 3s penalty every time you take damage, while the main tricks that we used required to take damage. These 3s penalties are consistent with the game objective to go for damageless levels, which is rewarded by a star on the map. However this would make for a very different TAS. IGT is also weird because it only starts when you leave the starting zone.

**!! Table**

\*”Strategy” indicates whether the regular goal zone or an out-of-bounds one was reached.

\*RTA starts at Helirin control and ends at IGT stop (= when Kururin looks happy).

\*”Matt’s RTA” refers to the [https://www.youtube.com/watch?v=DXB5DTYUei4|2015 “cheat bot” partial run by Matt Shepcar]. Neither all levels were done nor all birds were saved, so we only referenced the times that were comparable.

\*”Ryuto’s RTA” refers to the [1307M|2009 movie by Ryuto].

||Level name||Author||Strategy||RTA||Matt’s RTA||Ryuto’s RTA||

|Training 1 |E-Sh4rk|regular| 1.27| 1.42| / |

|Training 2 |E-Sh4rk|regular| 1.57| 1.62| / |

|Training 3 |E-Sh4rk|regular| 2.28| 2.28| / |

|Training 4 |E-Sh4rk|regular| 1.30| 1.80| / |

|Training 5 |E-Sh4rk|regular| 2.00| 2.00| / |

|Grasslands 1 |E-Sh4rk|regular| 1.90| 2.48| 5.87|

|Grasslands 2 |E-Sh4rk|OOB | 1.80| 6.05|10.72|

|Grasslands 3 |E-Sh4rk|regular| 6.68| / |11.40|

|Ocean 1 |mohoc |OOB | 1.40| 4.58| 5.35|

|Ocean 2 |mohoc |regular| 2.60| 2.72| 8.93|

|Ocean 3 |mohoc |regular| 6.33| / |15.75|

|Jungle 1 |mohoc |OOB | 2.67| 4.88|10.55|

|Jungle 2 |mohoc |OOB | 1.67|10.75|10.33|

|Jungle 3 |mohoc |regular| 3.20| / |14.22|

|Cake Land 1 |mohoc |OOB | 1.88| 4.00|12.70|

|Cake Land 2 |mohoc |regular| 2.80| 3.40|25.93|

|Cake Land 3 |mohoc |regular| 9.40| / |19.90|

|Cave 1 |mohoc |regular| 0.70| 1.03| 8.67|

|Cave 2 |mohoc |OOB | 5.13| 6.80|16.03|

|Cave 3 |E-Sh4rk|OOB | 3.95| 6.02|15.30|

|Cloud Land 1 |mohoc |OOB | 3.82| 5.17| 6.97|

|Cloud Land 2 |mohoc |OOB | 2.98| 4.90| 7.97|

|Cloud Land 3 |mohoc |regular| 7.08| / |10.07|

|Star Land 1 |E-Sh4rk|regular| 2,48| 2.72| 3.87|

|Star Land 2 |mohoc |OOB | 2.23| 2.95| 3.20|

|Star Land 3 |mohoc |OOB | 5.97| / |11.62|

|Ice Land 1 |E-Sh4rk|regular| 3.63| 3.73| 6.27|

|Ice Land 2 |mohoc |regular| 0.92| 1.93| 8.32|

|Ice Land 3 |mohoc |OOB | 4.78| / |27.95|

|Machine Land 1|mohoc |OOB | 3.10| 5.55| 8.30|

|Machine Land 2|mohoc |OOB | 2.58|17.53|18.35|

|Machine Land 3|mohoc |regular|13.77| / |29.03|

|Ghost Castle 1|mohoc |OOB | 3.12| / |20.97|

|Ghost Castle 2|mohoc |regular| 1.65| / |41.15|

|Ghost Castle 3|mohoc |regular| 8.80| / |28.13|

## !!!Closing

**!!mohoc’s personal comments**

So, now is my time to bore you with my life and the story behind this TAS. […]

\_\_TODO: bore people\_\_

I would like to thank several people:

\*E-Sh4rk

\*ThunderAxe31

\*Matt Shpecar

\*ViGadeomes