The Knytt Stories TASer's Handbook

A Comprehensive Analysis of the Knytt Stories Engine with a Focus on Tool-Assisted Speedrunning

Gliperal

2018 November

Dedicated to ristos, DonDoli, and the Knytt Stories speedrunning community

Contents

1 Disclaimer

The following information has NOT been peer-reviewed. The majority is based off of observation and experimentation. Most of the important source code is not available, so any claims made about the structure of the code are made as a result of testing and refining various hypotheses.

2 Definition of Terms

Apex Frames The frames of a jump during which Juni is at the highest whole pixel position. The first such frame is apex frame 1, the frame after that is apex frame 2, and so on.
B:O Denotes the object in the O'th position of bank B.
Ceiling Glide Climbing while inside of a ceiling to obtain the highest horizontal movement possible in a wallswim. *Section 7.6.2*
Collision (1) (see Wall) (2) The act of stopping horizontal or vertical movement by coming into contact with a wall.
cpx Centipixel or centipixels: one one-hundredth of a pixel.
cpx/f Centipixels per frame.
Effective Gravity Gravity minus Jump Hold Height (if holding the jump key). Juni's net acceleration downward.
Event Cycle The sequence in which the game engine executes various areas of the game's code. *Section 6.4.3*
Flag Warp A special kind of Warp that will redirect one screen to another if certain flags or powerups are on.
Glitchy Corner Jump Jumping within three frames of sliding off a corner, allowing a jump that touches the ceiling of that corner to be possible. *Section 7.1*
Ground Cycle A three or four frame cycle that occurs with Juni's velocity and subpixel position, while standing on the ground, that influences her jump height and when she will fall off ledges. *Section 6.5.1*
Juni Proper name of the protagonist in a majority of Knytt Stories levels and a term used generally in this paper to describe her hitbox (the Player Position object). *Section 6.2*
Knytt Stories A game created by Nifflas, and the subject of this paper. *Section 3*
Knytt Stories Plus A modified version of Knytt Stories that adds features, patches some glitches, and introduces some new glitches. The movement is largely identical to vanilla Knytt Stories.
KS Abbreviation for Knytt Stories.
KS+ Abbreviation for Knytt Stories Plus.
Malconfigured Shift A Shift object (often invisible) designed to block access to an area, but which uses Type 0: Spot. *Section 8.4*
MMF2 Multimedia Fusion 2, the game/application development program in which Knytt Stories was created. *Section 5*
OOB Out of bounds. May refer to the wallswimming state or entering a void screen.
Overlap When the hitboxes of two objects share a common pixel.
PCC Player Climb Checker, a narrow object that floats in front of Juni to detect when she is touching a climbable wall. *Section 6.2*
PMO Platform Movement Object, an extension for MMF2 that forms the basis of Knytt Stories's movement engine. *Section 4.2*
px Pixel or pixels.
px/f Pixels per frame.
Save Spot Bank 0 Object 1. The Glowing white lights that Juni uses to save.
Screen Position Juni's position in whole pixels, using the coordinate system established by the MMF2 engine, and used for all collision calculations. *Section 6.3.2*

Screen Transition A two frame pause during which one screen is cleared and a new screen is loaded, triggered by Juni leaving the edge of the play area or using a Shift. *Section 7.2*
Screen Transition Boost A small gain in height obtained by jumping twice in quick succession at a screen transition. *Section 7.2.1*
Shift Bank 0 Objects 14, 15, and 16. An object that can be used to teleport Juni around. *Section 7.4*
Subpixel Position A value in centipixels, typically between 0 and 100, used by the PMO for positional accuracy beyond the level of whole pixels. *Section 6.3.2*
Tile A 24 pixel square. These tiles are aligned to a grid and make up the Knytt Stories graphics.
Transitional Position The position Juni obtains on a screen after activating a Shift, but before the next screen has loaded. *Section 7.4*
Umbrella Dip Time spent in freefall between umbrella pumps. *Section 8.2*
Umbrella Float Time spent with the umbrella deployed, during which Juni's x-velocity is capped at 260 and y-velocity at 80 (before gravity).
Umbrella Pump Deploying and then retracting umbrella while in freefall (usually for only 2-3 frames) in order to dampen falling speeds. May occassionally refer to a very short umbrella dip during an umbrella float (usually in non-TASing circumstances). *Section 8.2*
Void Any screen in a Knytt Stories level that does not have a corresponding entry in the Map.bin file and defaults to displaying a purple background with the word "VOID" repeating across it.
Wall Any solid object with which Juni can collide, including, but not limited to, walls, ceilings, and floors.
Wall Cycle A two-frame cycle that occurs with Juni's velocity and subpixel position, while pressing against a vertical wall, that influences her walljump speed and how long it will take her to begin horizontal movement again. *Section 6.5.3*
Wallswim The state during which Juni is constantly overlapping a wall even after a PMO update. *Section 7.6*
Warps Bank 0 Object 20. An object used to tell the game which screen to load next after certain screen transitions. *Section 7.3*
X-Velocity A variable stored by the PMO and used to keep track of the speed of Juni's horizontal movement.
Y-Velocity A variable stored by the PMO and used to keep track of the speed of Juni's vertical movement.

3 Introduction

Knytt Stories is a 2007 platforming game developed by Swedish game developer Nicklas "Nifflas" Nygren. In recent years Knytt Stories has developed a small following in the speedrunning community – that is, people who attempt to beat the game as fast as possible. I recently took it upon myself to create a TAS (tool-assisted speedrun) of five of the main levels. The difference between a tool-assisted versus a regular speedrun is that TASes are expected to be perfect. The authors of a TAS will meticulously analyze every factor of the movement engine to determine the absolute fastest possible route to the frame. The inputs for a TAS are recorded frame by frame, so that they can be refined, whereas ordinary speedruns may be subject to human error. Over the course of my experimentation with Knytt Stories and writing the TAS, I came to be familiar with very minute details of the game engine. I also developed and proved a couple general principles of Knytt Stories TASing, that I could then apply easily without fear of there being a faster option.

This paper represents the summary of my work. It aims to explain every aspect of the game engine in detail, including glitches and speed tech. Ideally, it should contain enough information that aspiring TASers will be able to know exactly what to do before even testing anything in game. For the casual reader (or even TASer), there is no obligation to read it in order. Some sections may reference each other, and the core movement ideas are relevant pretty much everywhere, but if you are only interested in certain aspects, feel free to jump around.

4 Knytt Stories Source Code

The programming that went into Knytt Stories can be attributed to three sources: Multimedia Fusion 2, the Platform Movement Object, and Nifflas's work itself. We will cover each of these three elements in turn.

4.1 Multimedia Fusion 2

Fusion is a game/application/software creation program developed by Clickteam. Knytt Stories was created in the 2007 equivalent, Multimedia Fusion, or, more specifically, Multimedia Fusion 2 Developer (hereafter known as MMF2). In facilitating game development, MMF2 relies heavily on pre-encoded behaviors, meaning that even though Knytt Stories is open source, not all of its workings are known to us. Moreover, the movement of the main character herself is all handled by an extension to the MMF2 code, known as the Platform Movement Object.

4.2 The Platform Movement Object

The Platform Movement Object, or PMO, is an extension for MMF2 developed by Olle Fredriksson. It is an easy way to create platform movement, and more versatile than MMF's built-in platform movement. Much like MMF2, the source code itself is inaccessible, so we have to obtain most of our information through experimentation. Additionally, it is not without bugs, as we shall see. A large part of this analysis will be focused on analyzing the movement, glitches, and nuances of this object.

4.3 Nifflas's Code and Synergy

The order in which the code performs certain events is of key importance in how they will interact with each other. While we have access to all of the Nifflas–written code, so we know its order for sure, the rest of this information had to be extrapolated via experimentation. The order of events (up to cyclic permutation) is as follows:

Platform Movement Update	The Platform Movement Object takes control of Juni and moves her around according to the established physics.
Event List (Nifflas)	
Death Check	Detect whether or not Juni is overlapping something dangerous and initiate a death as necessary.
Map Scroll	Detect if Juni has entered a different screen, and queue a screen update as necessary.
KBD	The KBD object handles movement input. This part of the event list converts keyboard / controller input into common KBD values which the PMO can later read.
Umbrella	If umbrella has just been opened or closed, then update the PMO values to reflect umbrella physics or regular physics respectively.

Movement	Read the values from the KBD and convert them into corresponding PMO values. Note that although this will update Juni's velocity, it will not be applied to her position until the next PMO update. More on the specific order of these events in section 6.4.3.
Sounds	Play movement sounds when applicable, specifically: jumping, double jumping, landing on the ground, running along the ground, and climbing walls.
Animation	Play correct movement animations.
Hologram	Handle the creation and destruction of holograms.
Screen Render	Perform the aforementioned screen update in Map Scroll. Clear out existing objects and load in the ones for the new screen.
Object Behaviors (Nifflas)	MMF2 allows one to assign code to specific objects. In Knytt Stories, the most important of these behaviors are enemy movements and Shift object activation. It is important to note that this is only half of the code involved in a Shift (the other half being in Map Scroll in the Event List). More on that in section 7.4.
MMF2 Render	Perform the update that will actually write graphics to the application window. Although Juni and other objects may move around a lot during the events above, only their position/animation when we reach this section of the code will be rendered.
Debugger Pause	Pausing the application using MMF2's built-in debugger can only happen once per cycle and it will occur here. This is important to know when using it for TASing purposes.
Debugger Update	All the values in the debugger are updated. Note that this update occurs after the pause. This can also be considered to mean that the debugger update happens at the beginning of the above cycle. The result is that any values in the debugger will actually represent the values from one frame earlier. This would be troublesome for TASing, were it not possible to force an update a number of ways: click on a value, toggle the show/hide, etc.
Fast Loops	There are fast loops present in various areas of the Event List and Object Behaviors. Fast Loops consist of two parts: the trigger and the loop code itself. Unlike the rest of this table, the location of the loop code is irrelevant with regards to run order. Wherever it is defined doesn't matter, more like functions in actual programming languages. Once the loop is triggered it will run that section of code instantly, as many times as required, before continuing with the usual order of events. There are many circumstances in which an Object Behavior will run a Fast Loop written in the Event List.

Table 1: Order of event sections in the KS source code.

Notice that the Platform Movement Object updates at the beginning of each cycle, before the KBD can read input on that frame and before any values can be updated as a result. It follows that movement actions tied to button presses will take one additional update cycle (one additional frame) to register. For instance, Juni will continue to stand on the ground (or fall through the air) for one frame after pressing S but before jumping (or double jumping). Likewise with opening/closing umbrella or pressing A to walk.

Collecting powerups also obey a similar behavior, although the powerup collection occurs over a two-step process involving the powerup Object Behavior, and then the Event List. First cycle, Juni will walk into and collect the powerup, thereby triggering its Object Behavior to update Juni's powerup status. Second cycle, the PMO values will be updated during the Event List, but not yet affect her movement (thanks to the order of events). Third cycle, the collection of the powerup will begin to affect Juni's movement. This similarly means that we cannot take advantage of a powerup (such as double jump or umbrella) until one frame after collecting it, and we will not notice a change in movement until two frames after.

Conversely, the MMF2 render operations happen after the input in the update cycle, so events such as playing jump sounds, changing animations, and creating double jump smoke will all occur on the same frame as the corresponding button presses.

5 MMF2

5.1 General Information

First thing's first, Knytt Stories runs at 50 frames per second, or one frame every 0.02 seconds. This is not to be confused with the "storyboard frames" that make up the different parts of the game, or the "animation frames" that make up most of the sprite animations.

There are 7 storyboard frames in the Knytt Stories storyboard. Init, which occurs at the beginning and verifies the existence of files. Main Main. Select Level. Cutscene, which handles all cutscenes, including the Intro and Ending of a level. Gameplay, which will be our core focus. Install level. And Critical Error. The TASing of Main Menu, Select Level, and Cutscene are trivial, but there are two small optimizations worth mentioning.

The first optimization is for Select Level. By deleting all unecessary levels from the Worlds folder, the desired level appears on the first page, which saves time browsing for it. The second optimization is in the Cutscene frame. Cutscenes offers us two ways to advance the scenes, clicking on the right arrow button and pressing the right arrow key. Each of these actions has a cooldown, because consecutive presses of the arrow key or mouse button will count as holding them down. However, we can still advance optimally by interleaving the two, so that we click when the arrow key is released and vice versa.

Also crucial to mention is that changing between frames (particularly between Cutscene and Gameplay) will force that frame to refresh all of its content. This will become important in section 6.3.1.

5.2 Animation Speeds

Virtually all of the objects in Knytt Stories that use some kind of animation (with the possible exception of custom objects) rely on MMF2's built-in animation system. Because the specific animation frames play a key part in some interactions, it will be useful to explain MMF2's animation speeds. A single MMF2 animation speed unit seems to correspond to 1/100ths of animation frame per game frame. Thus, an animation with speed 100 would play one frame of animation for every game frame, i.e. 50 frames per second. An animation speed of 50 would require two frames to pass in game before advancing to the next frame of the animation. In short, an animation speed of s will cause each frame of animation to last for $100/s$ game frames. As we will see with Juni and subpixel positions, the rounding carries over. An animation speed of 30 would cause one animation frame per 3.33 game frames. Realistically this would result in the first animation frame lasting 3 game frames, the second lasting 4 game frames, then 3, 3, 4, 3, 3, 4, and so on.

5.3 Bouncing Ball Movement

Most of the objects which use a bouncing ball movement are enemies and their projectiles. This section may therefore be omitted on a first read, or until reading appendix A.

The MMF2 engine comes with a built-in movement for objects called the Bouncing Ball movement. Nifflas only uses it for directional movement, so we shall only discuss the relevant features of this movement type. At its core, the Bouncing Ball movement is vector-based –

that is to say, it takes a direction and a speed, and moves the object every frame according to those two parameters.

A speed of 8 corresponds to approximately 1 pixel per frame (when speaking about an object moving along one of the four cardinal directions). A speed of 16 corresponds to 2 px/f, and so on. As with Juni, enemies can only travel in whole pixel increments. Thus, an enemy with speed 12 will travel alternately 1 pixel and 2 pixels every other frame. As with the Every events, the bouncing ball movement is not based off an exact frame count. Unlike Every events, it doesn't seem to be affected by system lag, suggesting that it isn't based off a system clock either. Regardless, it is worth understanding that there may be subtle differences in position under otherwise identical circumstances. It seems unlikely that this variance will allow extreme positional changes, but it may be worth knowing when a couple pixels mean the difference between life and death.

The Bouncing Ball movement has exactly 32 possible directions, shown in Fig. 1. Direction 0 is due right, and the subsequent directions travel counter-clockwise, as with polar coordinates. Every one unit of direction corresponds to $\pi/16$ radians, so direction 0 and direction 32 are the same. It is possible to assign objects directions greater than 31 or less than 0, but they will simply be shifted by a multiple of 32 to return to the 0-31 range. The x and y components of the speed are calculated based on the direction value[1]. Because there are only 32 possible directions, projectiles that are designed to be shot straight at Juni can often end up missing because she is not actually standing directly in line with any of the 32 options. Note that this only applies to projectiles that move in a straight line. Gravity-affected projectiles cannot aim in the direction of Juni.

5.4 Glitches

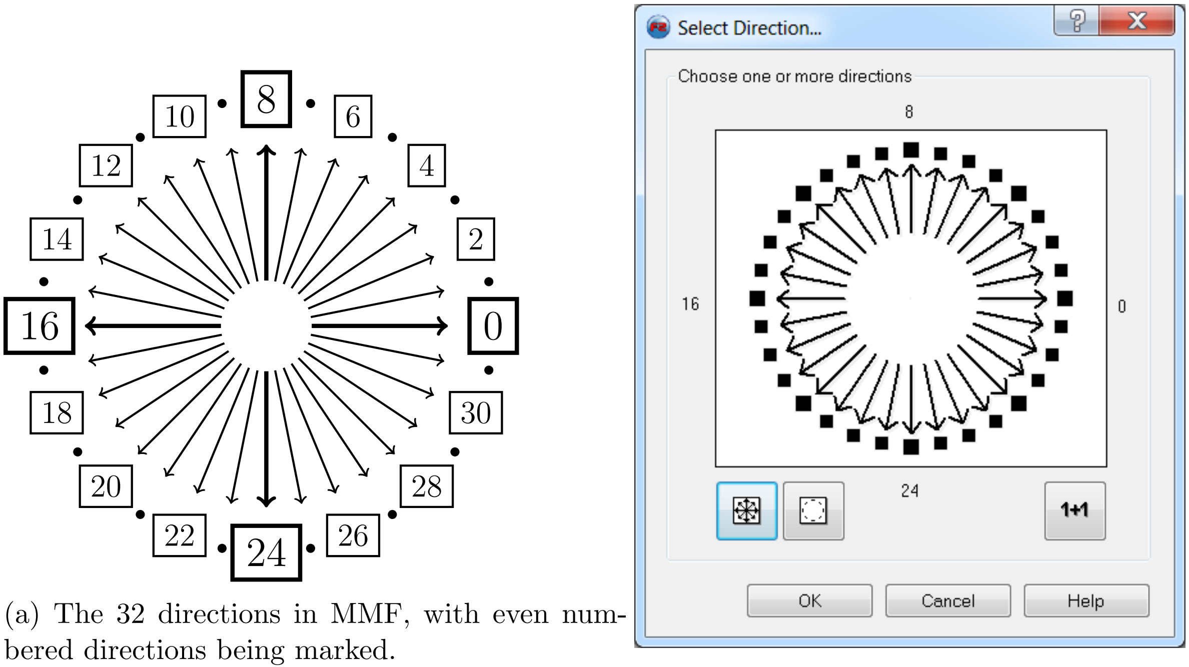
Of the three areas mentioned in section 4, the Multimedia Fusion 2 code itself is the strongest, but there are still a select few glitches. The most significant of these relates to timing.

There are certain events in MMF2 which require a duration in seconds. For unknown reasons, the MMF2 engine uses system time to calculate when that time has expired. This means that when Knytt Stories experiences slowdown it can cause the events to trigger faster than they would normally. There are two such timing events, each with unique repercussions. They are Every Events and Fade Transitions.

"Every" Events are a particular type of event in the MMF2 editor, where one can specify actions to occur "Every [some amount of time]." A couple of these events are related to cross-fading songs and ambiance, but the vast majority are inside of enemy behaviors. This means that certain enemy timings will be affected by game slowdown, sometimes causing them to attack more often, and sometimes causing them to change phase so often that they never have a chance to attack. Sadly, I have yet to find a level where this would be beneficial to a speedrun. Find out more specifics in appendix A.

Fade Transitions are a type of transition available in MMF2. In Knytt Stories, there are fade transitions when going from gameplay to a cutscene, as well as fade transitions when

[1] An object travelling in direction D with speed S will realistically have x-speed $S\cos(\frac{D\pi}{16})$ and y-speed $-S\sin(\frac{D\pi}{16})$. The y-speed is negative, because screen y values increase downwards, whereas Cartesian y-values increase upwards.



(a) The 32 directions in MMF, with even numbered directions being marked.

(b) The direction selector as it appears in the MMF2 engine.

Figure 1: MMF2 Directions

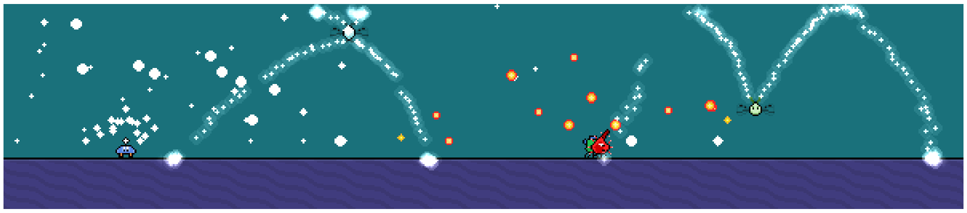


Figure 2: Enemy behavior under an instance of extreme game slowdown

certain objects are destroyed. Game slowdown will make these objects disappear faster relative to other game events. Additionally, pausing the debugger will not stop the fade timer, making these events occur faster than they would in real-time (instantaneously if you wait long enough). The list of objects with fade out transitions can be found in table 2.

Object	**Duration**	**Description and notable effects**
Save Spot flash	1.43s	It is not possible to save at another Save Spot until the flash from the first has been fully destroyed. Slowdown will therefore make it possible to save twice more quickly in game time.
Lock Block (15:12, 15:23, 15:24)	0.34s	Although slowdown will cause the lock block to disappear more quickly than usual, it has no effect on gameplay, since the lock block becomes non-solid as soon as it begins disappearing.

Pick Up Light	1.80s	The light that is emitted when collecting a powerup, after using a Shift, and when using the invincibility cheat.
Spring Light	1.10s	The light that is emitted when bouncing on a spring.
Holographic Player	0.22s	Possibly the fade with the biggest impact on gameplay (and especially TASing). When the hologram finishes fading out the Believed Position will return to the Player Position. This can cause different behaviors from otherwise 100% consistent enemies. It is important to keep in mind especially when using the debugger to TAS.

Table 2: Objects with fade out transitions

Note also that entering a different screen may appear to make the Save Spot flash and holographic player disappear, but really they have just turned invisible and one still has to wait for the fade to complete. This is the reason that it is not possible to immediately re-deploy a holographic player after having the previous one destroyed by a screen transition.

There is a rare glitch/corruption/we're-not-entirely-sure-what that can cause all of the fade transitions throughout the game to become instantaneous even if the game is playing at regular speed.[2] Given that it can persist after re-installing KS, it seems to be a glitch in the way the system time communicates with MMF2 applications, but that is speculative at best.

[2] https://clips.twitch.tv/BraveEmpathicKeyboardMVGame

6 Core Platforming Engine

6.1 Inputs

Before being used by the game, keyboard inputs are changed into input values that are stored in an object called KBD. The arrow keys are all converted into two values, Left-Right and Up-Down. When left and right are held at the same time, it is considered just holding right. When up and down are held at the same time, it is considered just holding down. There are also values for holding down the keys W, A, S, D, and Q. Note that the S key is only considered held down when Control is not (because Ctrl+S is a hotkey for toggling the game sound).

In addition, there are also "oneshot" values, which are used to record when a key is first hit instead of being held down. These are for S (jump), D (toggle umbrella), and Down Arrow (for use with hologram). Most of these events use the same code as holding down the buttons, plus the addition of a "Only one event when action loops." Due to this code structure, it makes it impossible to trigger the holding-a-key-down value without also triggering the key-oneshot.

Any actions which use the "Only one action while event loops" are also not possible to trigger on consecutive frames, by definition. There are four actions to which this applies. The most obvious is the "jump oneshot," which controls jumping and double jumping. For instance, it is not possible to double jump without releasing the jump key at least one frame prior. Other actions to which this applies are the "umbrella oneshot," "down oneshot" (which is only used for the double down hologram activation) and the hologram.

There is one way to circumvent this requirement. There are a couple times when the movement engine will be paused, usually during loading. When an action occurs on the first frame before and the first frame after this pause, then it is not considered a loop, and therefore will not be prohibited by the "Only one action while event loops" condition. The most notable situations are when changing screens (either through the side of a screen or via a Shift) where the game will pause for two frames to load the next screen. One instance in which this may be useful is doing umbrella pumps at a screen transition to gain height (see also section 8.2).

It's worth noting that the Knytt Stories Plus mod seems to treat screen transitions differently. Rather than allowing these kinds of buffered double inputs, it will ignore any inputs made during the two frames of a loading transition.

6.2 Anatomy of a Juni

There are 5 components that make up the plater character. The most relevant one to movement is the "Player Position." The Player Position hitbox is a rectangle 11 pixels wide by 17 pixels tall, which is used for collision detection with walls. The Player Position is also the hitbox used for detecting collision with enemies. Juni is the only animated sprite with this kind of hitbox – all the enemies and critters have hitboxes that perfectly match their animations.

The second component is the "Player Climb Checker," which, as its name suggests, checks whether or not Juni is capable of climbing / grabbing a wall. It is a 1 pixel wide by 9 pixel high rectangle that floats just in front of the player model (adjacent, but not overlapping). When the Player Climb Checker overlaps a wall or other solid object, then Juni is capable of climbing. It is 8 pixels shorter than the Player Position, which is distributed into 4 pixels on the top and 4 pixels on the bottom where Juni is able to collide with a wall but not able to grab it.

The third component is the “Believed Position.” It is a rectangle of the same dimensions as Juni’s hitbox, and serves as a representation for where the enemies think that Juni is (mostly for use with the hologram). When standing still, the two occupy the same region of the screen. When Juni is in motion, the Believed Position will lag one frame behind her. This means that during screen transitions and Shifts the Believed Position can be on the opposite side of the screen from Juni. No enemies are present during that frame, however, so the effects are negligible at best. Finally, when using Hologram, the Believed Position is set to the location of the hologram.

The fourth and fifth components are the only ones visible, and those are the “Player Character” which displays the animations and is what we would most commonly think of as Juni. For the most part it stays positioned over the Player Position object. The effects of this object are purely visual, and therefore not of great interest to TASing. Therefore, for the remainder of this analysis, “Juni” will refer to the Player Position unless explicitly specified otherwise. The last component, the “Holographic Player” is mostly visual, but also serves as a reference for where the Believed Position should rest when the hologram is deployed.

See Fig. 3 to see how all five components align with one another in typical gameplay.

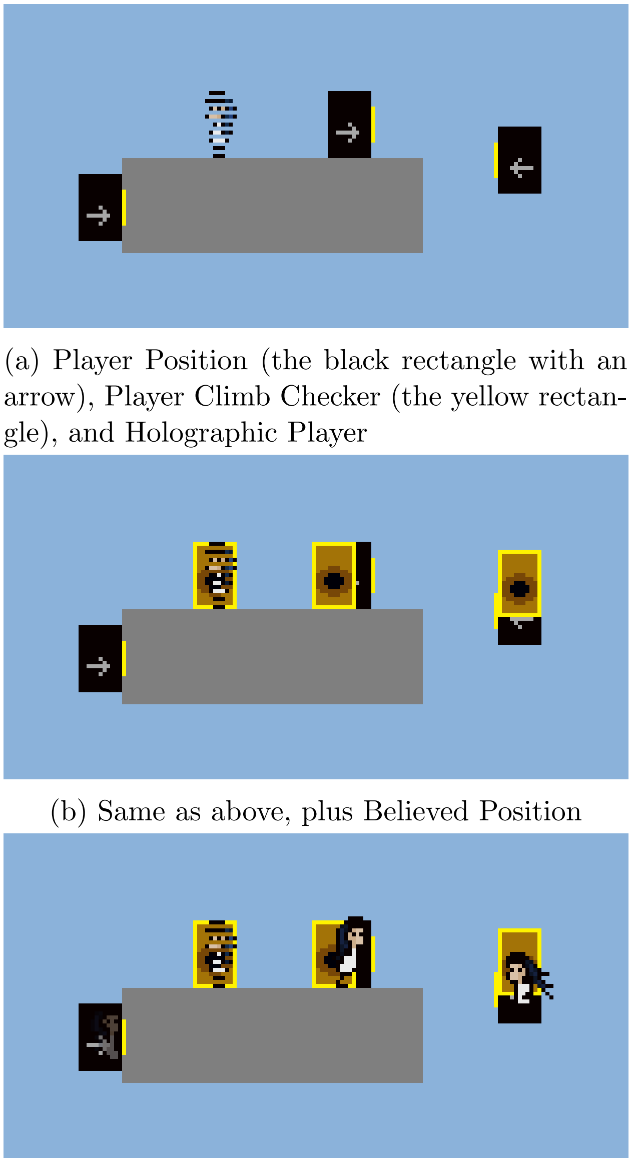
6.3 Position

6.3.1 Pixels and Centipixels

All positions in MMF, and therefore all collision detection, are calculated using whole pixels. Conversely, the Platform Movement Object stores all its information (positions, velocities, etc.) in centipixels, or one one-hundredths of a pixel. The difference between her position in centipixels and her position in whole pixels is referred to as a “subpixel position.” There are several notable effects of this discrepancy. The most obvious is that Juni will travel different distances although her speed is the same. With the run powerup, her horizontal velocity is 350 centipixels per frame (cpx/f), or 3.5 pixels per frame (px/f). When converted to whole pixels, this means that her hitbox travels sometimes 3 pixels and sometimes 4 pixels in an alternating pattern. When travelling at maximum walking speed (a speed of 180 cpx/f), the number of pixels she travels each frame will follow the repeating pattern 2, 2, 2, 2, 1.

Note that Juni’s subpixel position is NOT an actual position. The Platform Movement Object is not capable of setting Juni’s actual screen position (see next section). What it can do is move her in various directions and check for collisions. The subpixel position is a number between 0 and 100 that it uses to help determine how many whole pixels she needs to move. More detail on this in section 6.5.

It is important to take note of certain events that change Juni’s position, such as passing through a screen transition, using a Shift, and especially respawning from a Save Spot. Although these events will change Juni’s position on screen, they will NOT reset her subpixel position. The only way to reset Juni’s subpixel position is to collide with a wall or exit and



(a) Player Position (the black rectangle with an arrow), Player Climb Checker (the yellow rectangle), and Holographic Player

(b) Same as above, plus Believed Position

(c) Same as above, plus Player Character

Figure 3: From left to right in each image: Juni grabbing a wall (using hologram), hologram standing on the ground, Juni running horizontally, Juni falling vertically

re-enter the gameplay frame. The two ways of accomplishing the latter are by entering a cutscene or resetting the game.

6.3.2 Screen Position

The Knytt Stories screen consists of an array of pixels, 600 wide by 240 tall. As with Cartesian coordinates, Juni's position on this screen is described using a horizontal component, or x-position, and a vertical component, or y-position. The origin (x = 0, y = 0) is located in the top left corner of the screen, with positive x values travelling to the right, and positive y values travelling down. Thus, the bottom right corner is at (600, 240). Note that this is vertically mirrored compared to Cartesian coordinates.

All of the graphics in Knytt Stories are made from images called tilesets. These tilesets are broken up into square tiles 24 pixels wide by 24 pixels high. These tiles are then laid on a 24 pixel grid to create the graphics. The Knytt Stories window is 25 tiles wide by 10 tiles high. Because of the way these tilesets are used, level creators will frequently make solid ground using 24 pixel squares as well. This is almost standard practice, so we shall be referring to tiles often when discussing positional aspects.

It is important to note that Juni's origin (i.e. the location on the Player Position hitbox that corresponds to her position on the screen) is not at the top left corner. Rather, it is 5 pixels to the right and 10 pixels down from the top left corner of her hitbox. Thus, if Juni is at position 0,0, there are 5 pixels of her offscreen to the left, and 10 pixels offscreen on the top.

The data from tilesets (which make up the ground and scenery on a screen) is loaded in tiles 24 pixels wide by 24 pixels high. The game loads these tiles on a grid 25 tiles wide by 10 tiles high. Because of this natural division of the screen, it may often be useful to talk about positions in terms of whole tiles. For instance...

- When Juni is standing on a ground tile, her y-position will be equivalent to -7 (mod 24).

- When Juni's head is against a ceiling tile, her y-position will be equivalent to 10 (mod 24).

- When Juni is flush against the left side of a wall tile, her x-position will be equivalent to -6 (mod 24).

- When Juni is flush against the right side of a wall tile, her x-position will be equivalent to 5 (mod 24).

Note how these values derive naturally from the size of the Player Position, and the location of its origin.

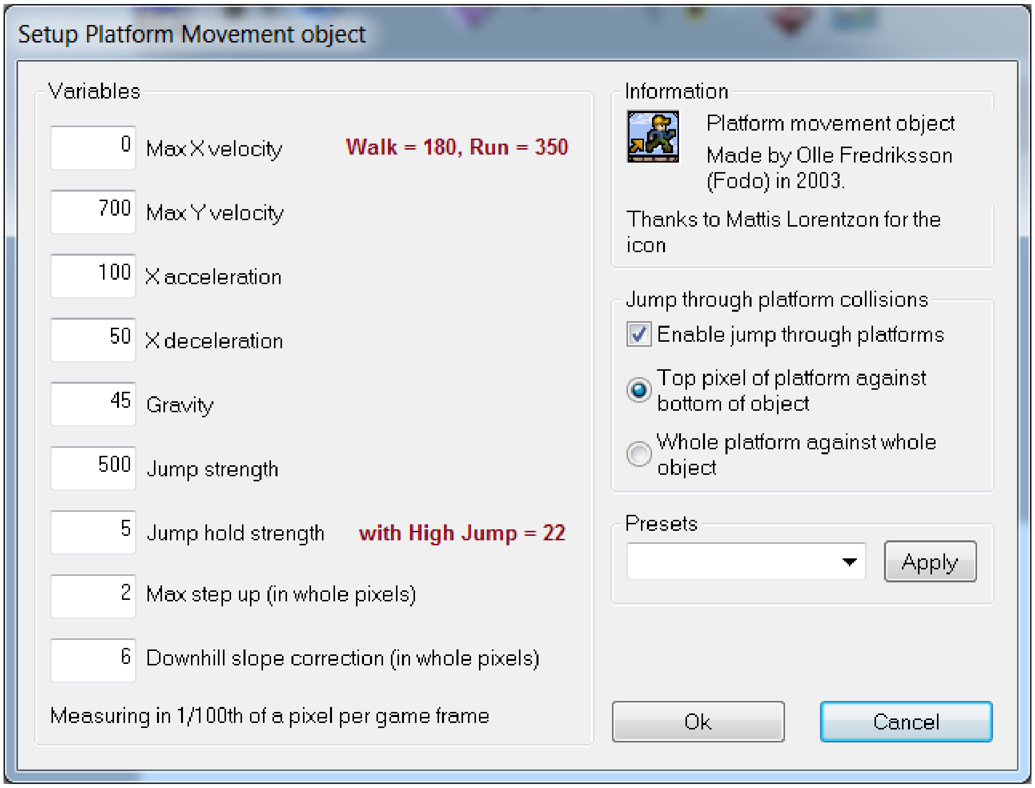
6.4 Movement Basics

6.4.1 Movement

The core of Juni's movement is quite simple, and relies only on a velocity and a wall-ejection system.

In addition to Juni's position and subpixel position, she also has a pair of velocities, a horizontal velocity and a vertical velocity. We will also refer to these as x-velocity and y-velocity, respectively. Just as her position is really stored in centipixels, these values are stored in centipixels per frame. Each frame, these two velocities are added to the two components of her position. Thus, a positive x-velocity means travelling to the right, and a positive y-velocity means travelling downwards. Because the velocities are applied in-between frames, one should avoid thinking of certain positions and certain velocities occurring "on the same frame."

6.4.2 PMO values



Setup Platform Movement object

Variables

0 Max X velocity — Walk = 180, Run = 350

700 Max Y velocity

100 X acceleration

50 X deceleration

45 Gravity

500 Jump strength

5 Jump hold strength — with High Jump = 22

2 Max step up (in whole pixels)

6 Downhill slope correction (in whole pixels)

Measuring in 1/100th of a pixel per game frame

Information

Platform movement object
Made by Olle Fredriksson
(Fodo) in 2003.

Thanks to Mattis Lorentzon for the icon

Jump through platform collisions

☑ Enable jump through platforms

● Top pixel of platform against bottom of object

○ Whole platform against whole object

Figure 4: Initial values for the Platform Movement Object

In Fig. 4 we can see the default values for the Platform Movement Object. The meaning and consequences of each one are as follows:

Item	Value	Description and notable effects
Max X Velocity	180 cpx/f (walk) 350 cpx/f (run)	Maximum value the magnitude of Juni's horizontal velocity is allowed to obtain. Values beyond this will cause the velocity to return to it's maximum value. For instance, with no run powerup, a horizontal velocity greater than 180 will become 180, and less than -180 will become -180. This value is also the fastest Juni can travel horizontally, ignoring Shifts and screen transitions. Under normal game conditions, this value is the speed of walking/running.
Max Y Velocity	700 cpx/f	Similar to the Max X Velocity, this is the limit on Juni's downward velocity. Upward velocity is irrelevant, because it is impossible for Juni to obtain a y-velocity lower than -700. Under normal game conditions, this value is terminal velocity.

X Acceleration	100 cpx/f^2	When holding a directional key, this is the acceleration that acts on the X component of velocity. When holding left arrow, 100 cpx are subtracted from x-velocity every frame. When holding right arrow, 100 cpx are added to x-velocity every frame. The Max X Velocity cap is applied after this addition.
X Deceleration	50 cpx/f^2	When not holding a directional key, this is the dampening acceleration that acts on the X component of velocity. If x-velocity is positive, then 50 cpx are subtracted every frame, and, if negative, then 50 cpx are added every frame. If this deceleration causes the magnitude of the x-velocity to reach 50 cpx or lower, then it will be set to 0.[3]
Gravity	45 cpx/f^2	Analogous to X Acceleration, this is the acceleration that acts on Juni's y-velocity. 45 cpx/f are added every frame, including when Juni is standing on the ground, inside a wallswim, and even climbing a wall. More on the effects of this in later sections.
Jump Strength	500 cpx/f	The vertical speed Juni gains when jumping. Realistically this value ends up being either -477 or -460, for reasons described in the subsequent section.
Jump Hold Strength	5 cpx/f^2 22 cpx/f^2 (high jump)	When holding the jump key (S), this value acts like Gravity but in reverse. When holding S with high jump, every frame, 45 cpx/f are added due to gravity and 5 cpx/f are subtracted due to Jump Hold Strength, resulting in an effective gravity of 40 cpx/f^2. With high jump, this effective gravity is 23 cpx/f^2. Like Gravity, this is applied almost universally when holding S, except when Juni's velocity is set explicitly after the jump hold strength is applied (see next section).
Max Step Up	2 px	The vertical distance Juni will attempt to "step up" during the horizontal component of her movement (see section 6.4.6)
Downhill Slope Correction	6 px 0 px (umbrella)	The distance which Juni will attempt to snap down to the ground when moving with a positive y-velocity. This snapping is designed to make her run smoothly downhill, rather than repeatedly bouncing down. This sort of behavior can still be seen when umbrella is deployed (and Downhill Slope Correction is set to 0).

[3] This 0 reset only applies when decelerating naturally, so x-velocity values between -50 and 50 cpx inclusive can still be obtained by decelerating with the opposite arrow key.

6.4.3 Order of Movement Events

The following are the events in the Movement section of the Event List. While they have a great deal of impact over Juni's velocity, the resulting velocities are not applied until the PMO section of the event cycle. Screenshots of the full movement code can be found in appendix B.

Left/Right Input	Record whether the user is holding left or right.
Run Powerup Check	Check whether or not Juni possesses the run powerup, and adjust her Max X Velocity accordingly (350 with, 180 without).
Jumps	Check whether or not to perform a double jump, wall jump, or regular jump, respectively. A jump consists of setting the y-velocity to -500 cpx/f. The order of these three doesn't really matter, because it is impossible for two of them to register at the same time.
Holding Jump Key	When holding the jump key, apply the Jump Hold Strength value. Unlike Left/Right Input, this is applied instantly, subtracting 5 or 22 from the y acceleration on this frame. Note that this occurs after the jump, so that even though the Jump Strength is 500 cpx/f, the starting y-velocity for jumps will actually be -477 with high jump, and -460 without.
High Jump Check	Check whether or not Juni possesses the high jump powerup, and adjust her Jump Hold Height accordingly (22 cpx/f with, 5 cpx/f without).
Wall Stick/Climb	If the Player Climb Checker is overlapping a wall, then adjust Juni's y-velocity accordingly: -250 if climbing (holding up arrow), 1 if sliding. Note that these events occur after the Holding Jump Key events, meaning that they will overwrite any changes. This is the reason that holding S has no effect when on a wall. Also set the "Was Recently Climbing" variable to 5 if we are climbing upwards.
Pull Over Edge	Like Normal Jump Countdown, decrement the Was Recently Climbing value by 1 every frame, including the first. If Was Recently Climbing has not yet reached 0, and we have left the wall, then set x-velocity to -300 or 300 (depending if we are facing left or right respectively). This Pull Over acceleration is designed to pull us over the top of a wall, but it will occur in other instances as well, including: climbing up a wall and then moving away from the wall; climbing down below a platform in a wallswim (more on this in section 7.6); and being teleported away while climbing a wall.

Table 4: Order of movement events in the movement section of the event list.

6.4.4 More about Jumps

There are certain circumstances in which the Jump Hold Strength will not be applied, specifically if Juni's y-velocity is explicitly set later than Holding Jump Key in the event cycle. Events that will do this include being on a wall and bouncing off of a spring[4]. Events that explicitly set Juni's y-velocity and still apply the Jump Hold Strength include jumping (causing jumps to start with either -477 or -460) and umbrella speed caps.

The Platform Movement Object considers Juni to be standing on the ground if two conditions hold true: (1) Juni has just landed on the ground (the PMO has performed a successful ground collision check from above) and (2) she has not moved vertically since. Juni is considered to be on a wall when the Player Climb Checker is overlapping solid ground. When Juni is standing on the ground or on a wall, the game initializes a "Normal Jump Countdown" to 5. This value decrements by 1 every frame (including the first one, so we can also consider it starting at 4). When it reaches 0, all jumps are considered double jumps. This is the reason we can still "wall jump" (technically a regular jump) up to 3 frames after leaving the wall, as well as 3 frames after running off a ledge: the Normal Jump Countdown is 4 while we are on the ground, it becomes 3 the first frame we leave the ground/wall, and the last frame we can jump is when it is 1, three frames after leaving the ground/wall.

The PMO movement will pause during screen transitions. If Juni was standing on the ground before a screen transition, then she will not move during the screen transition, satisfying both conditions to be considered standing on the ground. The Player Climb Checker overlap will likewise pause during screen transitions. It follows that if we teleport from a Shift while standing on the ground (or have been standing on the ground in the last three frames), to a destination in midair, then it will be possible to perform a midair jump at the destination. The same applies to dying while standing on the ground and then respawning from a midair save spot. In fact, because the inputs are paused during respawns, one need only hold the jump key any time during the respawn to buffer a first-frame respawn jump. More on the quirks of jumps and screen transitions in section 7.2.1.

6.4.5 Powerup Collection

Recall from section 4.3 that the event list occurs before the object behaviors in between render cycles. Because powerup collection is handled by the respective powerup behaviors, and the use of those powerups is handled in the event list[5], it is not possible to use a powerup on the same frame is collected. For powerups which require an input (double jump, umbrella, and hologram), the earliest that input can be applied is for the frame after the powerup is collected.

The earliest we can perform input is not necessarily the earliest we will see it reflected in Juni's movement, however. Because the PMO movement comes before the event list in the update cycle, movement powerups require one more frame, on top of the input delay, before the movement takes effect. This includes Run, Climb, Double Jump, High Jump, and Umbrella.

In addition to the input delay and the movement delay, High Jump will take one more additional frame to take effect, because of the order of the Holding Jump Key and High Jump Check events. When the Holding Jump Key is performed, the Jump Hold Height has not yet been updated and it will take one more frame for it to actually use the correct value.

[4] The spring occurs in an object behavior, which is after the event list.

[5] The one exception to this is keys, whose uses are handled by the respective lock behaviors. Consequentially, they are the only powerups that can act on the same frame they are collected.

Note that this does not apply to the run powerup. Although the Run Powerup Check does change Juni's Max X Velocity after the input is read, the Left/Right Input does not actually attempt to move Juni (and therefore apply the max velocity) until the PMO section of the event cycle.

Powerup	**Number of frames lost due to:**			**Total**
	Handling in Event List	Movement in PMO	Backwards event order	
Run	1	1	0	2
Climb	1	1	0	2
Double Jump	1	1	0	2
High Jump	1	1	1	3
Umbrella	1	1	0	2
Hologram	1	0	0	1
Keys	0	0	0	0

Table 5: Frames between the collection of a powerup and the first effects on gameplay.

These same delays apply to the removal of powerups as well (due to ShiftFlagOff Shifts). Consequentially, we are able to use most powerups (with the exception of keys) for a frame or two after travelling through a Shift that removes said powerup. This is particularly noticeable with double jump, as well as Shifts that simultaneously remove and deny hologram. Performing such a glitch with hologram will often leave the Believed Position object overlapping the Shift, causing it to trigger on every frame.

6.4.6 Collisions

Applying velocity on a given frame happens one component at a time. First the PMO will add the horizontal velocity, then the vertical velocity. This order is important to keep in mind for TASing, because it determines how close it is possible to cut corners. For each of these components, the movement is applied one pixel at a time. If one of these movements causes Juni to overlap a wall or other solid object, then the PMO will attempt to eject her from it. If Juni is moving horizontally, then it will attempt "step up." If the step up fails, then Juni is considered to have collided with a wall. Her horizontal velocity is set back to 0, and she is put one pixel back (theoretically right next to the wall).

During step-up, the PMO will attempt to move Juni up to 2 pixels vertically to put her up on a step. This is what allows her to run smoothly over curved surfaces and some steep slopes.[6] This step up procedure will occur even while Juni is moving upwards, and it will not alter her y-velocity.

The vertical velocity is applied in a similar fashion, one pixel at a time, checking for collision. If she collides with the ground, her y-velocity becomes 0. However, after the vertical movement is applied, if Juni has not already reached collision but she is moving downwards, it will perform "downhill slope correction."

[6] Because movement occurs one pixel at a time and this "step up" check happens each pixel, it is possible for Juni to run up slopes as steep as 2 (twice as tall as they are wide) if each pixel is a 2-pixel step up. We can see that this is an identical and opposite vertical speed to falling, since we move 3.5 px/f horizontally, translating to 7 px/f vertically on average.

During downhill slope correction, the PMO will continue to move Juni an additional 6 pixels downwards in search of the ground. If it finds a collision, then Juni will return one pixel, effectively making her snap to ground level. This means the maximum effect of downhill slope correction is actually 5 pixels. Much like the "step up," this snap will not actually alter her y-velocity, and the next frame she will continue to fall. If this happens near to a corner it will result in an edgebug (see section 7.7). If no collision is detected, then she returns to where she would have been without it. Use of the umbrella will also negate this effect, because it sets downhill slope correction to 0.

Downhill slope correction is what makes it possible for Juni to run smoothly down slopes.[7] Without it, Juni would attempt to run horizontally off of downward slopes, causing her to repeatedly run off and then fall back down, resulting in a jittery movement. You can see this effect by running with umbrella on certain slopes.

Note that collision detection only occurs as a result of attempted PMO movement. If Juni ends up inside the wall via other means (Shifts, screen transitions, etc.) no ejection will occur until the next time the platform movement object runs. By that time Juni is likely already inside the wall in a state known as wallswimming (see section 7.6).

6.5 Subpixel Anomalies

Recall that Juni's subpixel position is not an actual position, but a number between 0 and 100. This number represents an offset in the direction in which she is travelling. It is important that this not be confused with an offset from the top left corner. The fact that the subpixel position is relative to Juni's direction of motion means that changing direction will cause the subpixel position to change reference points as well. If Juni's subpixel position is very close to the top edge of a pixel at the peak of her jump, the very next frame it may suddenly become very close to the bottom edge of that pixel, as a displacement going up becomes a displacement going down.

It is unknown whether or not this subpixel position is between 0 and 100, or 0 and -100, or relative to the direction she is going, or relative to the direction she is coming. The pseudocode in Fig. 5 exemplifies one possible way in which the subpixel position may be handled each frame. In this case we have chosen Juni's subpixel position to stay between 1 and 100 inclusive, in the direction in which she is travelling. For the sake of simplicity, this is what we shall assume the PMO uses for the rest of the discussion[8].

Note that when Juni collides with a solid object, in addition to being set back, her subpixel position is set to -100. This is 100 lower than the usual range (between 1 and 100). The result of this negative value is a dampening effect on the motion, because the -100 centipixels must be covered before Juni can begin to move normally again. In fact, Juni must traverse 200 centipixels (2 whole pixels) before subpixel > 100 is satisfied and her screen position is modified. Moreover, it doesn't matter which direction Juni is moving

[7] With 180 cpx/f walking speed, the steepest slope it is possible to descend while walking is 5 px over 180 cpx, or about 2.777 rise/run. The steepest slope it is possible to descend by inching forward one pixel at a time is 5 over 1. If we allow Juni to be falling (edgebugging) during this, it becomes 12 over 1 when falling at terminal velocity.

[8] An interactive version for y-position is available online at `http://www.ksruns.com/resources/subpixelPosition_interactive.html`

```
int subpixel                    // Juni's subpixel y position
int velocity                    // Juni's current y-velocity in cp/f

...                             // Code related to updating movement values

subpixel += abs(velocity)       // Add velocity in the direction of motion
int pixelsToMove = 0            // How many whole pixels to move
while (subpixel > 100)
{
        subpixel -= 100         // Take 100 cpx from the subpixel position
        pixelsToMove += 1       // And add 1 px to the actual position
}

if (velocity < 0)               // Receive the directional information
        direction = ``upward''  //      from the sign of the velocity
else
        direction = ``downward''

try                                     // Move that many whole pixels
        move(pixelsToMove, direction)   // on the actual screen
catch (collision)                       // If we hit a floor or ceiling,
        subpixel = -100                 // set the subpixel position to -100
```

Figure 5: Centipixel Pseudocode

during this dampening period, because of the relativity of the subpixel position.

6.5.1 Ground Cycles

In Knytt Stories, Juni is never truly standing on the ground. Even after performing a successful ground collision, Juni will continue to be in a state of freefall. Because gravity is less than a full pixel per frame, and her subpixel position is set to -100, it will take a while for her to fall far enough for her position in whole pixels to update. As a result, the ground collision check is not performed every frame, leading to some strange effects.

When Juni first lands on the ground or hits a ceiling (or respawns from a Save Spot) her y-velocity is set to 0. After having collided with the ground or ceiling (but not after having respawned) her vertical subpixel position is also reset, to -100. As such, Juni is able to fall 200 centipixels before having to perform another collision check. Looking at her y-velocities (without holding S), she will travel 3 frames within this 200 cpx limit (0 cpx, 45 cpx, and 90 cpx). On the fourth frame her y-velocity will be 135 cpx, for a total vertical displacement of 270 cpx. This will put her subpixel position at 170, which is greater than 100, and moves Juni one pixel down. The game will perform another ground collision check. If she is still over the ground, then her y-velocity will be reset to 0, and the cycle will begin again. If not, then she will continue to fall downwards.

This behavior results in cycles where Juni can and cannot fall off the ledge. These cycles are 3-frames long under most circumstances, but they can become 4-frames long by holding S with high jump. This will make the effective gravity 23 cpx/f, and the first four frames will cover $0 + 23 + 46 + 69 = 138$ cenitpixels, resulting in a subpixel position of 38, which is small enough to not perform another ground collision check. The only way to obtain a ground cycle greater than 4 is by standing against a wall, resulting in a max y-velocity of 46 cpx/f (see section 7.1). Then it is possible to have 5 frames where the subpixel change is less than 200 $(0 + 45 + 46 + 46 + 46)$. This sort of ground cycle is most relevant during a wallswim. By manipulating these cycles so that the ground collision check occurs either on the last frame of ground or one frame later, we can induce Juni to either fall off of a ledge as late as possible or as early as possible, respectively.

These cycles also have an influence on jumping. If we jump on the first frame after a ground collision check, then Juni's subpixel position will be -100. Even though this subpixel position is supposed to be 100 cpx above the ground, the PMO always assumes that it is in the direction of motion. In this case, our direction of motion is now upwards, because of the jump, resulting in the -100 counting negatively towards our jump height. With high jump, our initial velocity will be -477, putting our updated subpixel position at 377, which becomes a subpixel position of 77 and 3 whole pixels of movement. Even though the jump started with a velocity of almost 5 pixels per frame, the first frame Juni will only travel 3 pixels.

In fact, if we wait until Juni is a bit later in the ground cycle (subpixel position between 24 and 100), then Juni will indeed travel the full 5 pixels on the first frame of her jump, because the subpixel position will count positively.

6.5.2 Max Jumps

We have just discussed how to use ground cycles to maximize (or minimize) jump height of a single jump. If fact, a fully extended jump can reach three different max heights[9] depending on the ground cycle. Ignoring subpixel anomalies, we would assume that the maximum height for a double jump is obtained by double jumping the frame before the velocity from the first jump becomes positive again. With High Jump, this means jumping and holding S for 20 frames, letting go for one frame, and then double jumping. This achieves a maximum vertical displacement of 103 pixels, but we can do better.

During a max height single jump with High Jump, Juni will actually stay at the apex of the jump (52px) for 4 frames. Double jumping right before a positive velocity means letting go on apex frame 2 and jumping on apex frame 3. As it happens, this gives Juni a subpixel position of 10 before the second jump. If we instead let go of S on the fourth apex frame and jump the frame after, then Juni will still be performing a double jump from the same screen position, but now her subpixel position will have increased to 79 from falling the additional 2 frames. Because the subpixel position is directionless, this gives Juni an additional 69 subpixels to her second jump, and allows her to reach a max height of 104 pixels.

This 104 pixel double jump requires a subpixel position of at least 14 before the double jump, allowing for some leeway in the inputs required to obtain such a jump. The subpixel position required without high jump is 21, also allowing some options for the maximum 58

[9] 27, 28, or 29 pixels without high jump; 50, 51, or 52 pixels with high jump.

pixel double jump. One interesting note about the 104 pixel high jump double jump is that it allows Juni to touch ceilings that are 5 tiles above the ground (5 tiles = 120 px. 104 px jump + 17 px Juni height = 121 px). Note that she can only overlap it with one pixel, which is not enough to grab a wall, but is enough to hit WIN tiles, collect powerups[10], etc.

The subpixel anomalies are less useful for max distance jumps, because we want to hold S as much as possible to maintain a low velocity, and because it only applies when we are changing direction. There are, however, rare circumstances in which it is useful. In the high jump double jump mentioned above (holding S for 22 frames before the double jump), we can let go of S for one frame when she is close to reaching the apex of her jump (specifically, after 18 frames of holding S in the double jump). This causes her negative velocities towards the end of the jump to be smaller, but not so small that we fail to reach the max 104 pixels. Because the values are smaller, however, it takes longer for her subpixel position to increase. The result is that we get a tiny bit farther jump just near the apex. It takes 1 frame longer to reach the max height, and we stay there for one frame longer than we would otherwise. This is the extent of the usefulness, however. The next two height changes occur the same as they would normally. After that the higher velocities (from letting go of S) start to catch up, and cause Juni to fall faster than she would have. As such, this technique is restricted to situations in which we have to squeeze out an extra pixel near the peak of a jump, and even then it is not always applicable.

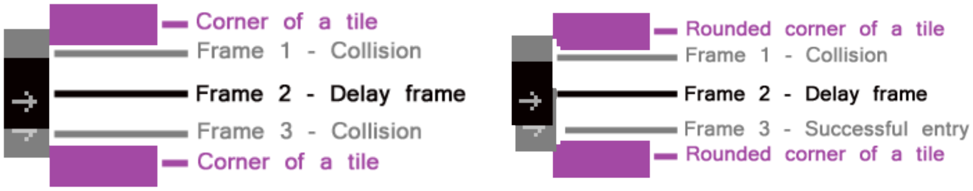
6.5.3 Wall Cycles and One Tile Corridors

Just as colliding with the floor or ceiling will result in a dampening of Juni's movement, so too will colliding with a wall horizontally. Because x-acceleration is 100 centipixels per frame, it requires at least two frames (100 + 180 or 100 + 200) before Juni is able to move again. When holding a directional key against a wall, this will cause cycles of one frame of movement and one frame of pause. This means that attempting to move away from a wall after colliding with it will also have one frame of delay. These are important to keep in mind for TASing, including when Juni climbs over the top of a wall or slides below it.

As far as casual gameplay is concerned, these two-frame wall cycles can also make it possible for Juni to completely avoid entering corridors that are one-tile high when sliding down a wall. Because Juni is 17 pixels high, and one tile is 24 pixels high, there are 8 y-positions where Juni can successfully enter the corridor. Since the maximum downwards velocity is 7 pixels per frame, it is possible for her to only spend on frame in this 8 pixel window. If that frame happens to be a delay frame, then Juni will not move horizontally, and therefore slide right past the corridor.

To avoid this problem (as a level designer, for instance) requires a minimum of 14 positions where Juni can enter the corridor. This guarantees at least two frames where Juni will be able to enter (one in the first 7, and one in the second), which guarantees in turn that Juni will enter the corridor regardless of wall cycle. In order for there to be 14 pixels where Juni can enter requires a corridor height of 30 pixels, or 6 pixels more than a standard tile. Examples of a sometimes impassable corridor and always passable corridor are in Fig. 6.

[10] An example of this can be seen in the first challenge of Xeia by Chironex.



(a) Regular 24-pixel corridor

(b) Curved 30-pixel corridor

Figure 6: Avoiding bad wall cycles on one-tile corridors

7 More Complicated Movement

7.1 Walls and the Climb Sphere

When the Player Climb Checker overlaps a wall or other solid object, it toggles the "Overlap" flag. Irrespective of the state of this flag, the regular platform movement will still continue to operate, meaning Juni will fall and collide with objects as usual. Keeping this in mind, the behavior of Juni when she is on a wall can be understood as small extensions to her usual movement.

When Overlap is true, and not holding the down arrow, any y-velocities greater than 1 will be set back to 1 (sticking to a wall). Realistically, this happens before the application of gravity, resulting in an effective wall stick speed of 46 cpx/f. When holding up arrow, the cut-off is -250 instead (wall climb). Similarly, this becomes an effective wall climb speed of 205 cpx/f. Neither of these are particularly useful in a TAS, except in very specific situations, because jumping from a wall gives a much greater speed and occurs on the same frame that the wall climb would take effect. For that reason, TASes will usually opt to jump up walls, rather than climbing. One effect that is worth noting, however, is the Pull Over Edge.

Pull Over Edge occurs when Juni was climbing up to 3 frames ago (in a similar way to the Normal Jump Countdown), and the Overlap flag gets toggled off. Her x-velocity gets set to 300 cpx/f in whichever direction she is facing. This will nudge her onto the top of a platform after finishing a climb. There are some other situations in which it will take effect, but note that in order to be considered climbing, Juni must not only be holding up next to a wall, but also breach the -250 speed limit[11]. It will trigger if Juni is teleported away from a wall. It will also trigger if Juni is climbing a wall and simply faces away from it, propelling her 300 cpx/f (350 if holding a direction, 250 if not) away from the wall. Moreover, this is not a one-time event, so the 300 cpx/f boost will occur for three frames after leaving the wall. During this time, Juni can also change directions, resulting in the fastest turn-arounds possible in Knytt Stories (from -350 cpx to 350 cpx or vice-versa). Performing a wall jump will negate this effect, but performing a regular jump after leaving the wall will not.

Like the Normal Jump Countdown, Pull Over Edge will also pause during deaths. If Juni dies while climbing a wall, then the boost from Pull Over Edge will occur after respawning, pushing Juni several pixels in whichever direction she is facing. This is why she may sometimes appear to drift away from a Save Spot after respawning.

As far as jumping is concerned, being on a wall acts the same as standing on the ground, in that it will reset the Normal Jump Countdown. It is therefore possible to move away from a wall and then jump up to three frames later and perform a normal jump. This is the first kind of walljump. The second kind is when Juni is hanging on the wall when she performs the jump (i.e. the Overlap flag is still on and she is not standing on the ground). In this case, she will perform a normal jump, but her x-velocity will also be set to 300 cpx/f in the

[11] This means that Pull Over Edge will not occur if Juni travels over the corner with sufficient vertical velocity (i.e. from a jump). This also means it is not possible to make it occur sliding down the bottom corner of a platform. The best attempt would be to have a corner 5 pixels high with a ceiling above, so that Juni has one pixel of climbable surface while against the ceiling. However, it still takes 3 frames to travel down one pixel after hitting a ceiling (see section 6.5.1), which takes up all of the Was Recently Climbing countdown.

direction opposite to the direction she is facing, and the Normal Jump Countdown will reset to 0. Under normal circumstances, this causes her to jump away from the wall. By facing away from the wall on the same frame, however, it is possible to make this 300 cpx/f occur towards the wall. This is a way of negating the effect, allowing Juni to stay next to the wall and climb faster (see section 8.3).

Because Juni is able to jump up to three frames after having left a wall, it is possible to jump during Pull Over Edge. It is also possible to gain a high enough velocity sliding down a wall that Juni has moved completely beneath the corner in those three frames, meaning that she is able to perform a normal jump underneath the platform. This is known as a glitchy corner jump.

7.2 Screen Transitions

When Juni exits the side of a screen, it triggers a screen transition. Specifically, this occurs when $x < 0$ or $x > 600$, or when $y < 0$ or $y > 240$[12]. Juni's screen position is also offset by 600 px horizontally, 240 px vertically, or both, depending on which side(s) of the screen she crosses. During this portion of the event list, the map position is updated, depending on Warps and Flag Warps. If this update causes either MapX or MapY to change, then it will activate a Screen Render.

The Render occurs over two update cycles (two frames). The first cycle removes everything from the previous screen, loads the new background, and loads the background images and solid ground (layers 0-3). The second cycle loads all the objects for the next screen (layers 4-7). The reason Nifflas chose to break it up into two cycles is unclear, but likely because of issues with unloading old objects and loading new objects on the same frame (with no update to the core MMF section of the update cycle in between).

During these two frames of rendering, the Platform Movement Object is frozen, the Player Position remains in place on the new screen, and the velocities, subpixel position, etcetera do not update. As mentioned in section 6.1, the fact that the movement engine is paused but the game is still running allows us to press buttons on two "consecutive" frames, allowing for some movement techniques not otherwise possible, such as gaining height with umbrella pumps (more on this in section 8.2).

Because of these two frames of freeze time, it is optimal in TASing to avoid screen transitions where possible. One can also combine two screen transitions into one. If Juni crosses a side edge and a top or bottom edge on the same frame, then it will cause one 2-frame screen transition, instead of two totaling 4 frames.

7.2.1 Screen Transition Boosting

Ordinarily, when Juni leaves the ground, she has three frames to perform a jump. This is controlled by the Normal Jump Countdown (see section 6.4.4). If Juni leaves the ground by jumping, however, the Normal Jump Countdown is instantly set to 0, denying her of those three extra frames. This is to prevent Juni from jumping in midair three frames after

[12] This means there are actually 601 pixels horizontally and 241 pixels vertically where Juni can be on a screen, and also prevents any rapid screen transitioning abuse, because it requires 2 pixels to turn around at best.

jumping normally, giving her about a 13 pixel boost. As it turns out, this is still possible by jumping during one of the two pause frames of a screen transition.

A curious thing happens when the Platform Movement Object attempts to jump while paused. The y-velocity will be set correctly (in this case to -500), but the position will not yet change. It is likely because of this lack of movement that the PMO still considers Juni to be standing on the ground. This causes her to regain the Normal Jump Countdown on the next frame. Thus, when Juni does leave the screen transition, she will be travelling upwards as if having jumped, but still have 3 frames to perform a second jump. This is also possible when climbing off of screen and while running/climbing into a Shift. It is not possible with a double jump, because there is no such countdown for double jumps.

7.3 Warps (0:20)

Warps occur as part of the screen transition sequence, as soon as Juni has passed the bounds of the screen. If Juni passes through two screen transitions at once, it will activate the Warps for both. This can cause stacking of identical Warps[13]. For example, if Juni is supposed to travel 4 screens over regardless of which direction she exits the screen, then it is possible to make her travel 8 screens over by leaving at a corner. This is not applicable to Flag Warps, which are specific to the screen Juni enters, and do not depend on which boundary(s) she crosses.

It is not possible to collect a powerup on the same frame as a screen transition. If possible, it could potentially be used to abuse Flag Warps. The reason it is not possible is because the screen transition and subsequent unloading of objects (in this case the powerup) occurs in the event list, before the object behaviors that control collecting powerups. It is possible to activate a Warp the frame after using a Shift, however, resulting in the Warp and the Shift combining. More on this in section 7.4.

7.4 Shifts (0:14, 0:15, 0:16)

Shifts are the objects 14, 15, and 16 in bank 0. The fundamental behaviors of a Shift are the same regardless of which one of the three is used. Additionally, there are 4 types of Shifts, specified by the ShiftType key in a world.ini file. The different types of Shifts affect the size and shape of the Shift hitbox, but nothing else. Additionally, the hitboxes will remain the same even when invisible. This can often lead to Shifts that only partly cover a tile and which can therefore be bypassed (more on this in section 8.4). The four types of Shifts are as follows:

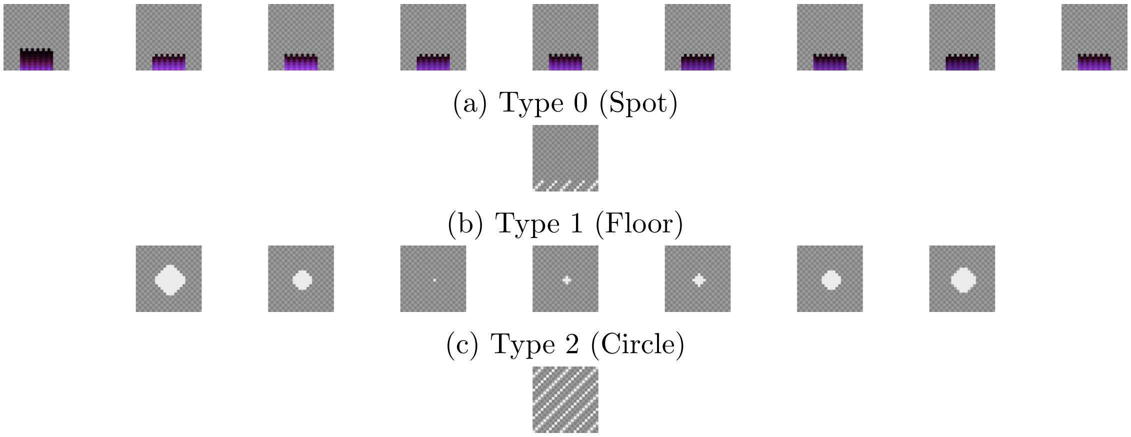
Type 0 (Spot): The default type of Shift, and also the one most commonly used. It is a purple, nearly-rectangular area on the floor, animated over 9 frames. The rectangle of the first frame is 12 px wide by 8 px high, and all subsequent frames are 12 px by 6 px. The speed of animation is set to 25, so each frame of animation lasts 4 frames of the game, or 0.08 seconds.

[13] The opposite is true on KS+, where only the X warp will activate if leaving the corner of a screen. This makes it possible to completely skip vertical warps on screens with multiple warp directions. For unknown reasons, this doesn't seem to apply when the X warp takes Juni to the same screen.

Type 1 (Floor): Roughly speaking, this is a rectangle a full tile (24 px) wide by 4 px high. This makes it shorter and wider than the default Shift. Visually and in terms of collisions, it is comprised of white diagonal lines 1 pixel wide and 4 pixels apart. As it is not perfectly rectangular, there are 6 pixels in the top left and 6 pixels in the bottom right where Juni can be within the 24 by 4 rectangle and not trigger the Shift.

Type 2 (Circle): This Shift is more similar to the flashing white circle from the original Knytt. Unlike the other three types, this one is elevated above the ground. It consists of 7 frames of animation (6 unique), each one a different sized near-circle, ranging from 1 pixel in diameter to 11 pixels. It is centered half a pixel down and right of the center of a tile. The animation speed for this one is 75, meaning every third frame of animation will occupy two frames of the game.

Type 3 (Square): The closest Shift to covering a full tile. It is comprised of the same diagonal white lines as type 1, and has the same property where 6 pixels in the top left and 6 pixels in the bottom right will permit Juni to overlap the tile, but not trigger the Shift.



(a) Type 0 (Spot)

(b) Type 1 (Floor)

(c) Type 2 (Circle)

(d) Type 3 (Square)

Figure 7: The four types of Shifts. A gray checkered background if included as reference for the containing tile.

Unlike Warps, Shifts are handled in object behaviors and do not pause the Platform Movement Object, making them far more exploitable. The process starts when a Shift is activated in the object behavior. It will update Juni's x-position and y-position, causing her screen position to update immediately. It will also update the MapX and MapY values. Assuming that the Shift does not stay on the same screen, this will signal to the screen transition code that a new render is required. Recall from section 4.3 that after the object behaviors are finished comes one more platform movement update before the update cycle reaches the event list again. Because the screen transition code (which also tells the PMO to freeze) is in the event list, Juni will continue to move for one more frame after the Shift is activated.

Note also that this movement occurs after the Shift object has moved Juni to a new position on (or potentially off) the screen. PMO collision detection will occur using this new position on the old screen. Let us refer to this position as the transitional position. The transitional position is not anywhere we would expect Juni to be during normal gameplay (it is neither the position where the Shift is activated, nor the position after fully teleporting), which can lead to some bizarre effects:

- If the transitional position is in midair, but the destination position is on the ground, then Juni can clip into the ground with sufficient downwards velocity.
- If the destination position is sufficiently close to solid collision, and the transitional position is unobstructed in that direction, then Juni can clip into it with enough velocity in that direction.
- If both the Shift and the destination are in midair, but the transitional position is within ground or another solid object, then Juni will act as though she landed on the ground, including being able to perform a normal jump for three frames.
- If the transitional position overlaps an enemy, Juni will NOT die, because of an After Teleport Countdown that is set in the object behavior.
- If the transitional position overlaps a powerup, Juni will collect it ONLY if it is a key. This is because the keys (0:21 - 0:24) have object behaviors that occur after the Shifts (0:14 - 0:16), whereas all the other powerups (0:3 - 0:10) occur before.
- Similarly, Save Spots, win tiles, no climb tiles, and sticky tiles (0:1, 0:2, 0:11, and 0:13) will not have an effect in the transitional position, but no jump tiles (0:25) will.
- If Juni crosses a screen transition the one frame she spends in the transitional position, it will activate any Warps on the starting screen, instead of Warps on the destination screen, which can place Juni in unintended places. Note that this is only possible for Shifts with ShiftQuantize set to false, because the quantized positions of regular Shifts are too far away from the edges of the screen.

All Shifts that have ShiftQuantize set to true (default) will center Juni on the tile to which she teleports, using the formula x-position $= 24a + 12$, where a is the index of the tile starting at 0. By default, quantize Shifts will also set her y-position directly to ground level ($24a + 17$). If a Shift has ShiftAbsoluteTarget set to true, however, then her y-position will instead by set to $24a + 12$. Unlike regular Shifts, this 5 pixels of extra height makes it possible to end up in a standard ceiling tile just above the destination of the Shift, but only if it is a ShiftAbsoluteTarget. Conversely, it makes it harder (albeit still possible) to enter a standard floor tile[14].

[14] Travelling down at max velocity of 7 px/f and teleporting to 5 px above the ground will place her only 2 px into the floor, one of which will be immediately lost to wallswimming. Rapid jumping is then required to lower her enough for the PlayerClimbChecker to take effect (see section 7.6). In fact, because she is only 1 pixel into the floor, attempting to move side to side would only cause the 2 px step-up to eject her from the ground.

7.5 Sticky, No Jump, and No Climb (0:13, 0:25, 0:11)

The no climb tile is object 11 in bank 0. If the Player Climb Checker overlaps a No Climb tile (or Juni does not possess the climb powerup), then the Player Climb Checker is moved to screen position -100, -100. Because no collision can make it there during normal gameplay, this effectively negates Juni's climb ability. If mods exist with the capability to create off-screen collision, then it may be possible to activate the Player Climb Checker overlap even without the powerup[15].

The No Climb tile has transparent areas, allowing there to be situations in which the Player Climb Checker is fully enclosed within the bounds of the no climb object, but not overlapping it. The orange area in Fig. 8 represents the top pixel of all such locations where this is possible.

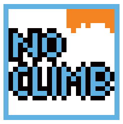


Figure 8: Climbable locations within a no climb tile.

The sticky block is object 13 in bank 0. Its collision can be considered a 24 pixel square. When created, a sticky is displaced one pixel upwards, so that it extends one pixel above full ground tiles. If Juni is overlapping a Sticky tile, it blocks the user from performing left or right inputs. Use of the walk key is not affected. If moving horizontally when sticky contact occurs, Juni will continue to move for a small amount while natural deceleration takes effect. Like no climb, there are holes in the sticky tile graphic, but the Player Position is too large for them to take effect, so it can be considered a 24 pixel square for all practical purposes.

The no jump tile is object 25 in bank 0. Visually, it functions identically to the Sticky tile, including the 1 pixel displacement upwards. When Juni overlaps (or continues to overlap) a no climb tile, it sets a Jump Block countdown to 6 (realistically 5, similar to the Normal Jump Countdown and Was Recently Climbing (see section 6.4.4)). When the Jump Block countdown is greater than 0, it prevents the effects of the jump button, namely jumping and decreasing the strength of gravity. Note that it does not block the input entirely, or else holding S and leaving a no jump area would cause a jump. Presumably this countdown was created to override the 5 frame Normal Jump Countdown. Realistically, a value of 3 would likely be sufficient[16]. A value of 6 makes it so that Juni is unable to jump or hold S for 5 frames after leaving the no jump tile.

[15] There are a couple ways in which this could occur. Moving pieces of collision would allow us to have collision that moves to -100, -100. Blocks that can be spawned at will could be spawned at negative coordinates. Solid custom objects can either be made big enough or shifted with an offset to extend past the screen and overlap -100, -100.

[16] A value of 2 is not, in the case of best-case ledge fall off cycles, where the PMO will still think Juni is on the ground (see section 6.5.1) after having left the bounds of the no jump tile. This would make it possible to jump off of the very edge of a no jump platform.

7.6 Wallswimming

Wallswimming is not a different state as far as the movement engine is concerned, but for our purposes it is useful to think of it as one. We shall consider Juni to be wallswimming when she is overlapping collision even after the PMO has performed an update.

7.6.1 Wallswim Movement

The core mechanics of Juni's movement are preserved in the wallswimming state. All jumping, climbing, updating of velocities, etcetera behave in the same way. The only differences occur when the PMO attempts to move her screen position. Because she is inside of a wall, every collision detection will be positive. Due to the way the PMO movement works, this will cause Juni to move 1 pixel opposite to the intended direction of movement.

When holding left or right, Juni obeys the same wall cycles described in section 6.5.3. This means that she will only perform a collision check every other frame, and therefore move 1 pixel backwards every other frame. The result is that holding left will cause Juni to travel right 1 pixel every two frames and vice-versa.

Even when Juni is inside of a wall, she will continue to fall. When the Player Climb Checker is also overlapping the wall, this will result in 5-frame ground cycles, which translates to moving up 1 pixel every 5 frames. It is possible for the Player Climb Checker to not overlap the wall if Juni is sufficiently close to the edge of a wall and facing outwards, or if she is only four pixels deep into a floor or ceiling. When this occurs (or while holding the Down Arrow), Juni will assume regular ground cycles. This will cause her to move up 1 pixel every 3 frames normally, or 4 frames when holding S with high jump.

When holding the up arrow, the ground cycles are overwritten, and Juni attempts to move up 205 cpx (see section 7.1). Recall from section 6.5 that it requires more than 200 centipixels of travel after a collision to move again. The 205 cpx/f from climbing makes this just barely possible[17], causing Juni to "climb" downwards at 1 pixel every frame.

Note that this downward climbing is only possible when the Player Climb Checker is overlapping the wall. The PCC floats in front of Juni, so being close enough to the side of the wall (or the side of the screen) and facing outwards is enough to make it so that Juni can no longer move downwards while climbing. Moveover, the PCC is not as tall as Juni. If she is only 4 pixels deep into a ceiling or floor, then the PCC will not overlap. If attempting to use a Shift to enter a wallswim, Juni must be able to get at least 5 pixels deep in order to easily maintain the wallswim via this downwards climb. Using a normal Shift, Juni must be able to obtain at least 5 px/f of downward momentum. Using a ShiftAbsoluteTarget, Juni cannot get deep enough in the floor with climbing alone, and must resort to other methods to maintain the wallswim. The four pixels where Juni can touch a ceiling and not climb are also worth noting, for they form a key component of the ceiling glide.

[17] If Nifflas had decided to make the climb strength even a little smaller (245 instead of 250), Juni would only be able to move down one pixel every other frame in a wallswim.

7.6.2 Ceiling Glide

Recall from section 6.2 that the Player Climb Checker is 4 pixels shorter than the Player Position on both the top and the bottom. This means that holding up will eventually cause the Player Climb Checker (but not Juni) to leave the wall. Because Juni was just climbing, this will trigger the three frames of Pull Over Edge (see section 7.1). These three frames will instantly set Juni's horizontal velocity to ±300 and attempt to move her forward, resulting in three frames of movement 1 pixel backwards. Moreover, because the Player Climb Checker overlap is no longer true, she will be unable to climb, and therefore begin to move back upward again. This will occur on a regular ground cycle.

Assuming not holding S, this results in 4-frame cycles, 3 from the ground cycle in the non-overlap position, plus one frame of inverted climbing in the overlap position. Of these four frames, 3 of them are moving 1 pixel backwards, making this kind of movement faster than holding a directional key, which will only move Juni 1 pixel every other frame. It is not possible to move 1 pixel on the fourth frame, because Juni's subpixel position has been reset, and only the instantaneous 300 cpx/f boost from Pull Over Edge can overcome it.

This movement is known as a "Ceiling Glide." It is also possible to use a screen transition, though the effects are slightly different. When Juni exits a wallswim by "climbing" off the bottom of the screen, then two things occur. First, Juni will come out on the other screen with a y-velocity of -205 cpx/f if she is holding the up arrow during the screen transition, because the game engine continues to run during that 2 frame pause. This upward velocity will cause Juni to re-enter the wallswim screen two frames later. The result is that Juni will rapidly oscillate between the two screens. During the two frames she is in the lower screen, the Push Over Edge from climbing will apply, moving her forwards. During the two frames she is in the upper screen, she will collide with the wall, moving her backwards.

If the user is holding a direction during the Push Over Edge, her speed will be 350 cpx/f, or 700 cpx over two frames. If not, then her speed will be 200 cpx/f, for 400 cpx over two frames. Recall that she just collided with the wall beforehand, however, so her subpixel position will start with -100, and she will only move 6 pixels or 3 pixels, respectively. Once she re-enters the wall, she will collide in that direction, moving her one pixel backwards, as with a normal wallswim. It is not possible to avoid this backwards step, due to how fast she is moving. The net displacement is therefore 2px or 5px over eight frames (two in each screen, and two for each screen transition), making it slightly faster than wallswimming when holding a directional input.

7.6.3 Entering/Exiting a Wallswim

There is no known way to enter a wallswim without having Juni either teleport into a wall, or a wall load on top of Juni. The ways in which this can happen include:

- Travelling through a screen transition into a wall (section 7.2)
- Using a Shift to teleport into a wall, possibly due to glitches (section 7.4)

Once Juni is no longer overlapping the collision, she will leave the wallswimming state. Similar to entering, this can occur using screen transitions or Shifts as well. Unlike entering a wallswim, Juni is not blocked by the transition from collision to non-collision, so she can freely exit a wallswim by moving until she has left the wall. For the most part, this requires Juni to travel all the way outside of the wall. If she is within 2 pixels of the top of a wall, however, any horizontal movement will eject her from the wall, due to the 2 pixel step-up (see section 6.4.6).

It is easy to leave a wallswim through a wall or floor. As we have just seen with the Ceiling Glide, however, holding up will not allow Juni to exit a wallswim via the ceiling, due to the size of the Player Climb Checker. When gliding against a screen transition, Juni simply has to release the up arrow before the transition to circumvent the climb up from the screen below. This is a one frame trick, since it requires holding up the frame before transition and releasing it the first transition frame, but it is easy to spam attempts by simply mashing the up arrow button. Exiting through a regular ceiling is more difficult, however.

It is possible to travel downwards inside a wallswim without climbing by making use of jumps instead. When jumping inside of a wallswim, Juni will attempt to move 3-5 pixels up, sending her 1 pixel down. Note that jumping is basically always possible, because Juni is either considered on a wall (if PCC overlap is true) or has just performed a ground collision check (if PCC overlap is false). Recall from section 6.1 that jumps can only be performed once every 2 frames. Thus, the fastest Juni can travel down by jumping is 1 pixel every 2 frames. This does make it possible to travel down and out of a ceiling.

Once every two frames is more than enough to overcome the natural drift upwards due to ground cycles. Recall that Juni's subpixel position is reset to -100 after every collision (including from jumping in a wallswim) and that the natural ground cycle in a wall takes 5 frames after that to move up again. Moving downwards via jumping requires pressing S once every 5 frames, or 10 times per second. Simply maintaining height (one pixel up, one pixel down) requires pressing S once every 10 frames, or 5 times per second.

Note that these frequencies become higher when the Player Climb Checker does not overlap a wall, causing 3-frame ground cycles. Then it requires one S press every 6 frames to maintain height (8.3 Hz) or one S press every 3 frames to move down (16.6 Hz) This makes actually using the jumping technique to exit a wallswim through a ceiling extremely challenging for non-TAS. It may help to attempt the technique while rapidly going back and forth through a screen transition, using the 2 frames of freeze time to make the mashing less severe.

7.7 Edgebugs

Edgebugs are a glitch that make it possible to land on a platform without impeding a downwards fall. I chose the name based off a similar glitch in the Source Engine, which will stop a downwards fall but without having registered as landing on a platform. Both glitches require the user to land on the very edge of the platform while moving away from it.

In the case of Knytt Stories, the glitch occurs when Juni would be within 5 pixels of the ground on one frame (causing downwards slope correction to snap her to that piece of ground) and then move away from the ground on the very next frame. She continues to fall as normal, because the downwards slope correction does not reset her y-velocity to 0. Apart from that, however, she retains all the effects of having landed on the ground.

Technique	**X Movement (px/f)**	**Y Movement (px/f)**
Holding left/right	±0.5	
PCC overlap true		−0.2
PCC overlap false		$-0.3\bar{3}$
Down Arrow		
One of the above, plus holding S with high jump		−0.25
Up arrow		+1
Ceiling Glide	±0.75	
Screen Transition Glide (no input)	±0.25	
Screen Transition Glide (holding left/right)	±0.625	
Rapid jumping		+0.5

Table 6: Speeds of wallswim movement techniques

After an edgebug, Juni is able to perform a normal jump for up to 3 frames (as if she had just run off a platform). With sufficient downwards velocity during the edgebug, this makes it possible to start a normal jump from much lower than would normally be possible. Juni will also regain her double jump from an edgebug.

Because an edgebug causes Juni to snap downwards up to 5 pixels, obtaining edgebugs can be desirable when falling downward and x-position is not important. They will move Juni lower without changing her y-velocity, allowing her to continue to fall at terminal velocity. Note that it is not necessarily optimal to edgebug as many times as possible. Because Juni's fall at max velocity is quantized on 7 pixel intervals, only one edgebug size is possible off a given ledge[18]. Thus, it may be better to use one 5-pixel edgebug, than 4 1-pixel edgebugs.

[18] The possible edgebugs are from 0 pixels to 5 pixels. If Juni is 6 pixels above the ground she will be unable to edgebug, and the next frame she will collide normally with the ground, leading to no edgebug either.

8 TASing Theory

8.1 Principle of Even Distribution

Suppose that we wish to find out when is the best time to double jump to cross a large gap, or how far we can travel with exactly 10 umbrella pumps. This section outlines a general rule that is useful in a variety of TASing circumstances. Roughly speaking, it states that the maximum height we can obtain over a fixed distance requires that the ending velocities of each jump or umbrella dip be as close together as possible. Section 8.1.1 provides a more rigorous definition, and also a proof, and may be skipped. Section 8.1.2 provides some real TASing examples where this principle can be useful.

8.1.1 Why it works

For simplicity, all distance units in this section are assumed to be cenitpixels and all time units are assumed to be frames unless otherwise stated. For the purposes of this discussion, it is useful to consider the sections between each jump or umbrella pump, which will always start with the same y-velocities:

Definition 8.1. (Components) Let a component be defined as an arithmetic sequence of y-velocities which Juni might obtain during freefall, of minimum length 0, where the common difference is an effective gravity g, and any velocities above 700 (terminal velocity) become 700. Let the component type be one of the following, depending on Juni's powerup state and the input resulting in the component:

Component type	**Component y-velocities**
A jump without high jump	$-460, -420, -380, -340, \dots$
A jump with high jump	$-477, -454, -431, -408, \dots$
After releasing umbrella with high jump	$126, 149, 172, 195, \dots$
After releasing umbrella without high jump	$143, 183, 223, 263, \dots$
Falling off an edge without high jump	$0, 40, 80, 120, \dots$
Falling off an edge with high jump	$0, 23, 46, 69, \dots$
Bouncing off a spring without high jump	$-655, -615, -575, -535, \dots$
Bouncing off a spring with high jump	$-655, -632, -609, -586, \dots$

Table 7: Component Types

It will also be useful to consider the last velocity in a component, and the velocity that would come next. Consider an arbitrary component c with velocities $c_1, c_2, \dots, c_n$. We shall use the subscript $c_\blacksquare$ to denote the last velocity (that is, c_n). We shall say that the "pending velocity" of c refers to the next term if the sequence of velocities were to continue (that is, c_{n+1}), and shall be denoted by the subscript $c_\triangle$.

Before moving on, it shall be useful to mention some properties of components. The following are easily verified:

Component Property 1: A component is uniquely determined by component type and number of frames.

Component Property 2: For any component a with effective gravity g, $a_{\blacksquare} + g \geq a_{\triangle}$.

Component Property 3: For any two components a and b of the same component type with effective gravity g, $a_{\triangle} > b_{\triangle} \implies a_{\blacksquare} \geq b_{\blacksquare} + g$.

Component Property 4: For any two components a and b of the same effective gravity, $a_{\blacksquare} \geq b_{\blacksquare} \implies a_{\triangle} \geq b_{\triangle}$.

Each component represents consecutive frames during which Juni is in freefall. We can then consider a sequence of these components to represent a combination of jumps, double jumps, umbrella pulls, etc. I have decided to call such a sequence of jumps a "combination":

Definition 8.2. (Combination) A Combination $\mathcal{C}$ is an ordered series of components $\mathcal{C}_1, \mathcal{C}_2, \ldots$ which share a common effective gravity. The "fall distance" of a combination is the sum of all the y-velocities across all the components. The "range" of a combination is the range of the pending velocities (that is, given $(\mathcal{C}_1)_{\triangle}, (\mathcal{C}_2)_{\triangle}, \ldots$, then the range of $\mathcal{C}$ is the difference between the two that are farthest apart).

Because of the usefulness of the minimum and maximum pending velocities in calculating range of a combination, it will be helpful to create some notation for them as well. Therefore, given a combination $\mathcal{C}$, say that $\mathcal{C}_{\text{min}}$ refers to the component that minimizes pending velocity over $\mathcal{C}$, and likewise say that $\mathcal{C}_{\text{max}}$ maximizes it. Then the range of $\mathcal{C}$ is simply $(\mathcal{C}_{\text{max}})_{\triangle} - (\mathcal{C}_{\text{min}})_{\triangle}$.

Finally, we can examine all possible such techniques for crossing the same gap. We will define the set of all such possibilities as a "combination space." Ultimately we would like to discover which of these techniques loses the least height at the end, which is where the Principle of Even Distribution comes in.

Definition 8.3. (Combination Space) Given a number of frames F, an effective gravity g, an integer N, and a list T of component types $t_1, t_2, \ldots, t_N$ such that the effective gravity of each component type is g. Then the corresponding Combination Space is the set of all combinations $\mathcal{C}$ that satisfy the following:

- The number of components in $\mathcal{C}$ is the same as the number of component types in T.
- Component $\mathcal{C}_n$ is of type t_n for all integers $n \in [1, N]$.
- The cumulative number of frames across all the components in $\mathcal{C}$ is F.

Lemma 8.1. *Given Combination Space X, and a combination $\mathcal{A} \in X$ that minimizes fall distance over X, then one of the following holds for every component $\mathcal{A}_k \in \mathcal{A}$:*

- *$\mathcal{A}_k$ has no frames.*
- $(\mathcal{A}_k)_{\blacksquare} \leq (\mathcal{A}_{min})_{\triangle}$

Proof. We shall prove that any component that does not satisfy the first criteria must satisfy the second. Suppose that arbitrary component $\mathcal{A}_k$ has at least one frame, and say that the component that minimizes pending velocity is at position m (that is, $\mathcal{A}_m = \mathcal{A}_{\min}$). Consider a combination $\mathcal{B}$ identical to $\mathcal{A}$ in every respect, but with one less frame in component k, and one more frame in component m. It is easy to show that this combination must also be in X. Because $\mathcal{A}$ minimizes fall distance over X, this new combination cannot have a smaller fall distance, and therefore the last velocity of $\mathcal{A}_k$ (which was removed in creating B) cannot be greater than the last velocity of $\mathcal{B}_m$ (which was added). Thus $(\mathcal{A}_k)_\blacksquare \leq (\mathcal{B}_m)_\blacksquare$. Since $\mathcal{B}_m$ is one frame increased from $\mathcal{A}_m$, we have $(\mathcal{B}_m)_\blacksquare = (\mathcal{A}_m)_\triangle$ and thus

$$(\mathcal{A}_k)_\blacksquare \leq (\mathcal{B}_m)_\blacksquare = (\mathcal{A}_m)_\triangle = (\mathcal{A}_{\min})_\triangle$$

which is what we wanted to show. □

Theorem 8.2 (Principle of Even Distribution)**.** *Given a Combination Space X, then a combination $\mathcal{A} \in X$ minimizes fall distance over X only if it minimizes range.*

Proof. If X is a combination space with total frames 0, then the proof is trivial. Therefore assume that X has a total frames of at least 1.

Let $\mathcal{A}$ be a combination that minimizes fall distance over X. Suppose, towards a contradiction, that $\mathcal{A}$ does not minimize range. Then there exists some other combination $\mathcal{B} \in X$ whose range is smaller than that of $\mathcal{A}$. Because $\mathcal{A}$ and $\mathcal{B}$ share the same component types and $\mathcal{A} \neq \mathcal{B}$, we can use Component Property 1 to show that there is at least one position n for which $\mathcal{A}_n$ and $\mathcal{B}_n$ do not have the same number of frames. Moreover, because $\mathcal{A}$ and $\mathcal{B}$ have the same number of total frames, there must be two such positions m and n, where $\mathcal{A}_m$ has more frames than $\mathcal{B}_m$, and $\mathcal{A}_n$ has less frames than $\mathcal{B}_n$.

Because $\mathcal{A}_m$ has more frames than $\mathcal{B}_m$, then either $(\mathcal{A}_m)_\triangle > (\mathcal{B}_m)_\triangle$ or $(\mathcal{A}_m)_\triangle = (\mathcal{B}_m)_\triangle = 700$. If the latter is true, then we can use the fact that $\mathcal{A}_m$ has at least one frame in Lemma 8.1 to show that $(\mathcal{A}_{\min})_\triangle \geq (\mathcal{A}_m)_\blacksquare$. Since $\mathcal{A}_m$ is at least a one frame increase of $\mathcal{B}_m$, we have $(\mathcal{A}_m)_\blacksquare \geq (\mathcal{B}_m)_\triangle = 700$, and therefore $(\mathcal{A}_{\min})_\triangle \geq 700$. Neither $(\mathcal{A}_{\min})_\triangle$ nor $(\mathcal{A}_{\max})_\triangle$ can be more than 700. This implies that the range of $\mathcal{A}$ is 0, meaning $\mathcal{B}$ cannot have a smaller range, which is a contradiction. Thus we can assume $(\mathcal{A}_m)_\triangle > (\mathcal{B}_m)_\triangle$.

Similarly, we have either $(\mathcal{A}_n)_\triangle < (\mathcal{B}_n)_\triangle$ or $(\mathcal{A}_n)_\triangle = (\mathcal{B}_n)_\triangle = 700$. If the latter is true, then $(\mathcal{A}_{\max})_\triangle = (\mathcal{B}_{\max})_\triangle = 700$. In order for the range of $\mathcal{B}$ to be smaller than that of $\mathcal{A}$, we must have $(\mathcal{A}_{\min})_\triangle < (\mathcal{B}_{\min})_\triangle$, implying that $\mathcal{A}_{\min}$ has a smaller pending velocity than its corresponding component in $\mathcal{B}$. Either way, there exists n such that $(\mathcal{A}_n)_\triangle < (\mathcal{B}_n)_\triangle$.

Because the range is the difference between the maximum and minimum pending velocities, this means that either the maximum pending velocity for $\mathcal{B}$ is smaller than that of $\mathcal{A}$, or the minimum pending velocity for $\mathcal{B}$ is larger than that of $\mathcal{A}$. We shall examine the two cases separately:

Case 1: $(\mathcal{B}_{\min})_\triangle > (\mathcal{A}_{\min})_\triangle$

Recall that there exist m for which $(\mathcal{A}_m)_\triangle > (\mathcal{B}_m)_\triangle$. Using Component Property 3, we can show that this implies $(\mathcal{A}_m)_\blacksquare \geq (\mathcal{B}_m)_\blacksquare + g$. From Component Property 2, we have $(\mathcal{B}_m)_\blacksquare + g \geq (\mathcal{B}_m)_\triangle$, and therefore:

$$(\mathcal{A}_m)_\blacksquare \geq (\mathcal{B}_m)_\triangle$$

Since $\mathcal{A}_m$ has at least one more frame than $\mathcal{B}_m$, we know that $\mathcal{A}_m$ has at least one frame, and therefore Lemma 8.1 applies, giving us:

$$(\mathcal{A}_{\min})_\triangle \geq (\mathcal{A}_m)_\blacksquare$$

Finally, since $(\mathcal{B}_{\min})_\triangle$ is the minimum of a set that includes $(\mathcal{B}_m)_\triangle$, we have $(\mathcal{B}_m)_\triangle \geq (\mathcal{B}_{\min})_\triangle$ and thus the following contradiction:

$$(\mathcal{A}_{\min})_\triangle \geq (\mathcal{A}_m)_\blacksquare \geq (\mathcal{B}_m)_\triangle \geq (\mathcal{B}_{\min})_\triangle > (\mathcal{A}_{\min})_\triangle$$

Case 2: $(\mathcal{B}_{\max})_\triangle < (\mathcal{A}_{\max})_\triangle$

Similar to case 1, we can use the fact that $(\mathcal{A}_n)_\triangle < (\mathcal{B}_n)_\triangle$ to show that:

$$(\mathcal{B}_n)_\blacksquare \geq (\mathcal{A}_n)_\triangle$$

Because $(\mathcal{A}_{\max})_\triangle > (\mathcal{B}_{\max})_\triangle \geq (\mathcal{B}_k)_\triangle \;\forall\; k$, we know that $\mathcal{A}_{\max}$ must have at least one more frame than the component in the same position in $\mathcal{B}$, meaning $\mathcal{A}_{\max}$ has at least one frame. Then we can use Lemma 8.1 to show that

$$(\mathcal{A}_{\min})_\triangle \geq (\mathcal{A}_{\max})_\blacksquare$$

Then using the fact that $(\mathcal{A}_{\min})_\triangle$ is less than or equal to $(\mathcal{A}_n)_\triangle$ by definition, we have

$$(\mathcal{B}_n)_\blacksquare \geq (\mathcal{A}_n)_\triangle \geq (\mathcal{A}_{\min})_\triangle \geq (\mathcal{A}_{\max})_\blacksquare$$

We can then use Component Property 4 on $(\mathcal{B}_n)_\blacksquare \geq (\mathcal{A}_{\max})_\blacksquare$ to obtain the contradition:

$$(\mathcal{B}_n)_\triangle \geq (\mathcal{A}_{\max})_\triangle$$

$$(\mathcal{B}_{\max})_\triangle \geq (\mathcal{B}_n)_\triangle \geq (\mathcal{A}_{\max})_\triangle > (\mathcal{B}_{\max})_\triangle$$

Because it is neither possible for $\mathcal{B}$ to obtain a greater minimum pending velocity or a lesser maximum pending velocity than $\mathcal{A}$, it follows that the range of $\mathcal{B}$ cannot be smaller than that of $\mathcal{A}$. Thus, $\mathcal{A}$ must minimize range, which concludes the proof. □

8.1.2 What it means

The Principle of Even Distribution makes a few key assumptions. First and foremost, it assumes that subpixel positions are absolute, whereas in Knytt Stories they are actually relative to the direction of motion (see section 6.5). When travelling always upwards (as is the case with quick double jumps), this does not make a difference. If, however, our y-velocity flips from negative to positive or vice-versa, we can expect up to 1 pixel of error. In a long double jump we can expect three such places, two at the apexes of the jumps, and one at the double jump itself. As such, we could potentially be off by 3 pixels. In many cases, the results obtained from the Principle of Even Distribution will still be the best scenario, but it is important to keep in mind that it is not an exact model.

The second assumption is that the effective gravity is constant. Assuming that we don't collect or lose high jump part way through a sequence, this is a reasonable assumption. In order to maximize height, we want to be constantly holding S. The only two exceptions would be to gain the benefit of a subpixel anomaly (but that is quite rare) or to let go of S in order to perform a double jump. Obviously we would only want to let go of S for one frame before the double jump. This single frame will add 45 to the y-velocity instead of 40 or 23. The very next frame will be the jump, resetting the velocity, so there is only one frame that is affected. Moreover, every combination that contains a double jump will have this one frame. We can therefore consider this to be a constant 5 cpx or 22 cpx addition to the final y-displacement predicted by the Principle of Even Distribution. Because this is the same for every combination, it does not change the results of the theorem.

Finally, it is worth pointing out that the order of the components don't matter. It was worth keeping them ordered in the previous section to make the proof easier, but realistically we can perform an umbrella pump before a double jump or a double jump before an umbrella pump, and the sum of the y-velocities will be the same. It may therefore be useful to re-arrange components in order to avoid obstacles, avoid screen transitions, or obtain better subpixel manipulation.

In order to describe the effects of the principle, let us use an example. Suppose that we have high jump, and we jump from the ground 35 pixels away from a wall. Where is the best place to time a double jump to maximize height when we hit the wall?

In this scenario, the combinations are made up of two components: the initial jump and the double jump. Assuming the wall is straight vertical and that we are moving at max speed, we have a fixed number of frames before hitting it. In this case, that number of frames is 35 px (3500 cpx) divided by our velocity (350 cpx/f with run), which gives us 10 frames. Using the results of the Principle of Even Distribution, we want the ending velocities of both components to be as close together as possible. In this case, giving 5 frames to each will do just that:

Initial Jump	Double Jump
-477	-477
-454	-454
-431	-431
-408	-408
-363	-385

Figure 9: Y-velocities during a 10-frame double jump combination

Note that without letting go of S, the last y-velocity in the initial jump would be -385, making it exactly the same as the last velocity in the double jump, and therefore perfectly evenly distributed. This may cause some jumps to look deceptively more evenly distributed than others (try adding up the y-velocities in Fig. 10 for each one, and you'll find that they are identical).

A more complicated example come from the Gustav's Daughter TAS. After collecting Hologram out of bounds, we need to travel roughly a screen and a half upwards. The fastest way to accomplish this is to: jump into the void, re-enter the wall to gain another jump, repeat as necessary. Because re-fueling at the wall takes up 4 frames of screen transition,

Initial Jump	Double Jump
-477	-477
-454	-454
-431	-431
-408	-408
-363	

(a) 5 frames, then 4 frames

Initial Jump	Double Jump
-477	-477
-454	-454
-431	-431
-386	-408
	-385

(b) 4 frames, then 5 frames

Figure 10: Y-velocities of two 9-frame combinations, both of which are evenly distributed.

plus an additional 4 frames just to wallswim back out of the wall[19], we want to minimize the number of jumps. It is possible to reach the top with 9 cycles (a wall embedding and a jump) that last 19 frames, and then we reach the top on the 14th frame of the next jump, for a total of 185 frames. If we apply the principle of even distribution to this, we can instead use 5 cycles of 19 frames each followed by 5 cycles of 18 frames each, which will give us the total 185 frames, but we will have travelled further vertically by evenly distributing the jumps. Indeed, by using this method we breach the top on the 16th frame of the last jump, for a total of 183 frames (a 2 frame improvement). Evenly distributing once more yields 3 19-frame cycles and 7 18-frame cycles, which also puts us at the top after 183 frames, but we are one pixel higher than before, and this it the optimal jump combination.

8.2 Umbrella Pumping

Umbrella pumping is a technique used for swiftly crossing gaps that are large enough to require the use of umbrella. Because having umbrella deployed drops our max x-velocity from 350 to 260, it is useful to have it deployed as little as possible. If umbrella x-velocity were the same as regular x-velocity, there would be no issue with keeping umbrella out any time the y-velocity would increase above 103. Such is the case when Juni does not possess the run powerup, so we shall assume use of Run for this section. Deploying umbrella for even the minimum possible time of two frames will instantly cap our y-velocity at 80 cpx/f (technically 103 or 120, because of the order of events, and assuming holding of S. For sake of concision, we shall assume Juni possesses High Jump and the y-velocity cap with umbrella is 103), but the same principles apply without High Jump. After retracting the umbrella, it will take several frames of freefall to reach terminal velocity once again. An umbrella pump refers to only deploying umbrella for a few frames before retracting it again. An umbrella dip refers to the intervals in between umbrella pumps, during which Juni is in freefall. By performing these pumps every so often, we can significantly lower our average y-velocity with only a slight drop in our average x-velocity.

Generally the goal of umbrella pumping will be to cross a gap using the minimum amount of umbrella. Every frame in umbrella will advance Juni 260 cpx horizontally, whereas every frame not in umbrella will advance her 350 cpx. It does not matter where these frames of umbrella occur during the crossing, nor how they are grouped together. When umbrella is

[19] Each pixel of movement requires two frames, and the minimum we can end up embedded in the wall is two pixels, due to the one pixel of screen transition buffer

opened, x-velocity is instantly capped at 260 cpx/f. When it is closed, the natural 100 cpx/f^2 acceleration is enough to take Juni instantly back to 350 cpx/f. The objective is therefore to find a sequence of umbrella pumps that provides enough height to cross the gap with the minimum number of frames in umbrella.

Before we discuss solving this minimization problem, it is important to note that we cannot activate the umbrella toggle on two consecutive frames (see section 6.1). This means that the minimum time an umbrella pump can last is 2 frames, and that the minimum interval between umbrella pumps is likewise 2 frames. In fact, when in possession of high jump, if we were able to pump every other frame (for instance, if there were two keys for umbrella), then Juni would obtain a higher trajectory (and therefore be able to travel farther) than simply floating with umbrella. This is because she accelerates faster horizontally out of an umbrella pump than vertically. Even 2-frame pumps provide a slight improvement over floating, but it is less than 1%. See Fig. 11.

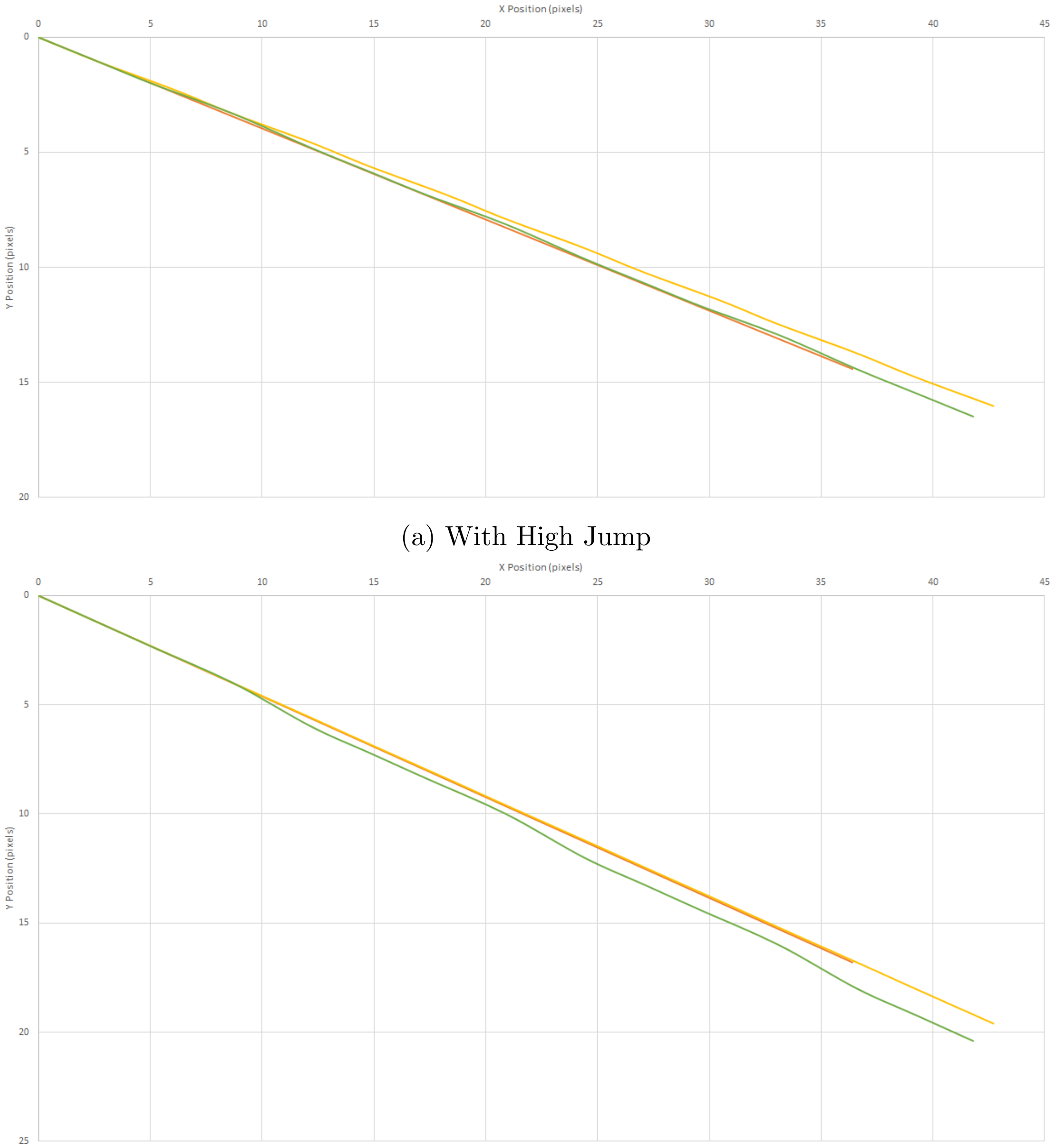
Ideally, we want most of the umbrella pumps to be 2 frames, possibly with one 3 frame pump if there are an odd number of umbrella frames. Note that it is possible to have a “1 frame” umbrella pump if it occurs at the very end and if Juni encounters a win tile, Shift, screen transition, etcetera, that would make the fact that she is still in umbrella for one more frame not count against her. This is a rare case, however, so we shall assume that all umbrella pumps are at least 2 frames. Using this information, we can show that mostly 2 frame umbrella pumps and possibly a 3 frame is the optimal situation.

Suppose that we have a situation in which Juni is using 4 or more frames in an umbrella pump, or two 3 frame umbrella pumps. Either way, we are able to split these into smaller pumps (the 4-frame becomes two 2 frames, and the two 3-frames becomes three 2-frames). This gives us an extra umbrella pump, and therefore an extra umbrella dip. If we split up a dip accordingly, the first half will stay the same, and the second half will improve, thereby providing us with a umbrella sequence that loses less height (see Fig. 12).

Note that this only applies in situations where we have not already reached the maximum number of umbrella pumps (that is, half of the frames in the jump with 103+ velocity are already spent pumping umbrella). If more than half of the frames are spent in umbrella, we can still divide them up into 2-frame umbrella pumps, if we add the additional stipulation that umbrella dips can be 0 frames. This is the equivalent of a four frame umbrella pump, and allows us to safely apply the same procedure.

We can therefore assume that all the umbrella pumps are 2 frames, with the last one being 2 or 3. Then, given the number of umbrella frames in a jump, we can calculate the number of umbrella pumps. It will simply be the number of umbrella frames divided by 2 (rounded down).

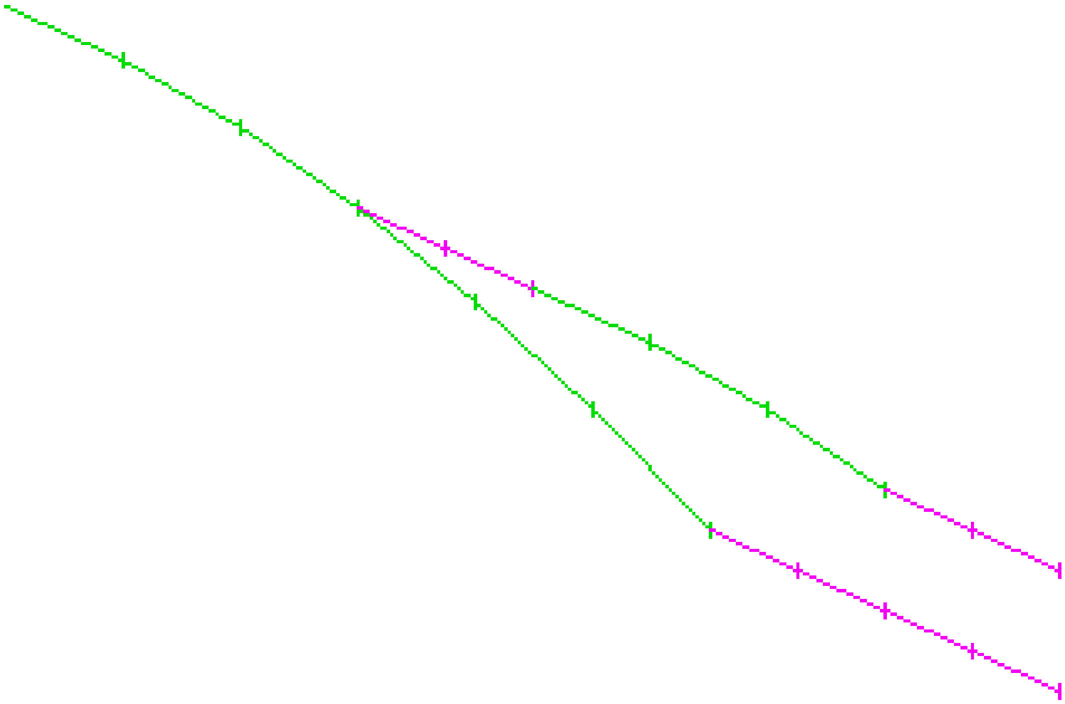
The principle of even distribution can be applied to umbrella pumping in the same way that it applies to jumps, but one extra condition must be imposed. The portions during which Juni has umbrella deployed her velocity will be a constant 260 horizontal and 103 vertical. We can therefore remove these constant intervals from the equation. The one additional condition that is required is that the number of umbrella pumps is fixed. This will ensure that the number of umbrella dips (the components in the theorem) is likewise constant. Then the application of the theorem goes as follows:



(a) With High Jump

(b) Without High Jump

Figure 11: Trajectories of umbrella pumping techniques over 15 frames. The red line represents simply floating, the yellow line represents 1-frame pumps (if they were possible), and the green line represents 2-frame pumps.



(a) Visual representation of trajectories

	X	Y
	350	160
	350	200
	350	240
	350	280*
	350	320*
	350	360*
☂	260	120
☂	260	120
☂	260	120
☂	260	120

(b) Velocities of the 4-frame pump sequence (in cpx/f)

	X	Y
	350	160
	350	200
	350	240
☂	260	120
☂	260	120
	350	160*
	350	200*
	350	240*
☂	260	120
☂	260	120

(c) Velocities of the 2-frame pump sequence (in cpx/f)

Figure 12: Separating a 4-frame umbrella pump into two 2-frame umbrella pumps (no high jump). Notice the difference between the y-velocities indicated with a *.

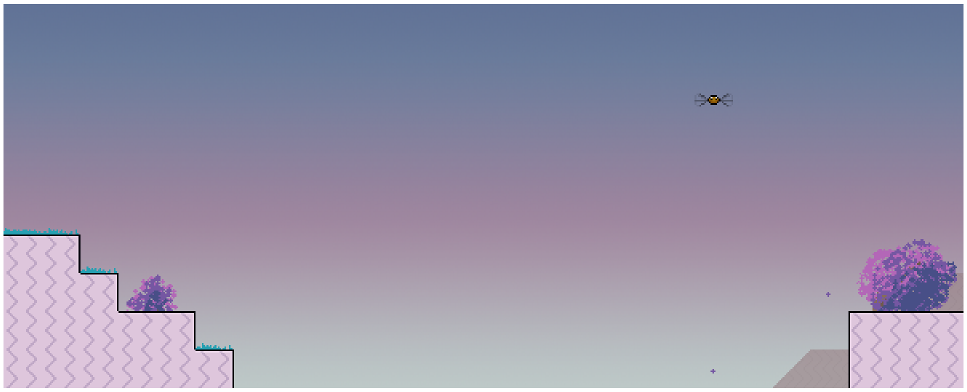


Figure 13: *Nifflas - A Strange Dream*, screen x971y998

- Preliminaries: Let our gap distance be x (in centipixels).
- Step 1: Fix a number of frames u which Juni will spend in umbrella.
- Step 2: The number of umbrella pumps / umbrella dips will be $N = \lfloor u/2 \rfloor$.
- Step 3: The horizontal distance travelled in umbrella (in cpx) will be $d_u = 260 \cdot u$.
- Step 4: The horizontal distance travelled in freefall (in cpx) will be $d_f = x - d_u$.
- Step 5: The number of frames spent in freefall will be $F = \lceil d_f/350 \rceil$.
- Step 6: Because N and F are constant for a fixed u, we can apply the principle of even distribution to the intervals between umbrella pumps, plus any preceding jumps and double jumps. This will give us the best possible combination for u frames of umbrella.
- Step 7: Repeat this process for all possible values of u, and then select the smallest one that still allows us to cross the gap.

In order to demonstrate this, we shall use screen x971y998 of Nifflas's *A Strange Dream* (Fig. 13). During the any% easy speedrun of this level, we cross a 17-tile gap in this screen from left to right. 17 tiles corresponds to 408 pixels. Because Juni is 11 pixels wide (and assuming she hits the final pixel of the left side corner), she must cross 399 pixels to reach the other side. x will therefore be 39900. Let us choose u to be 14. The rest of the values are then calculated as follows:

- Number of umbrella pumps $N = \lfloor 14/2 \rfloor = 7$
- Horizontal distance travelled in umbrella $d_u = 260 \cdot 14 = 3640$
- Horizontal distance travelled in freefall $d_f = 39900 - 3640 = 36260$
- Number of frames spent in freefall $F = \lceil 36260/350 \rceil = 104$

This leaves us with 104 frames Juni spends in freefall, which must be split up into 10 components (falling off the ledge, the first jump, the double jump, and seven umbrella dips).

Applying the principle of even distribution gives us the optimized solution shown in Fig. 14. Note that the layout used makes it so that reading top to bottom and left to right causes the y-velocities to be arranged in ascending order. This makes it easy to create such a table by hand, by starting at the top left and filling out y-velocities one at a time until the desired 104 frames has been reached.

Ledge Fall Off	**Initial Jump**	**Double Jump**	**Umbrella Pumps**						
	-477	-477							
	-454	-454							
	⋮	⋮							
	-40	-40							
	-17	-17							
0	6	6							
23	29	29							
46	52	52							
69	75	75							
92	98	98							
115	121	121	126	126	126	126	126	126	126
138	144	144	149	149	149	149	149	149	149
	167	167	172	172	172	172	172	172	172
	190	190	195	195	195	195	195	195	195
	213	213	218	218	218	218	218	218	218

Figure 14: Y-velocities during a double jump with seven umbrella pumps.

Note that the first column cannot go above 7 frames, or else our first jump will become a double jump. Also note that we have not yet included the additional +22 in the two velocities that occur immediately before both of the jumps. Taking this into account, we can calculate the final y-displacement by adding up all of the y-velocities in the above table, plus fourteen 103 y-velocities (during the umbrella frames), plus the 44 cpx correction for the two double jumps, yielding a final value of:

$$-1681 + 1442 + 44 = -195$$

Juni has travelled 195 centipixels (almost 2 pixels) upward from her initial location, and will therefore be able to make it up onto the ledge on the other side. In fact, Juni can make it up onto the ledge even after having travelled down two pixels below it, thanks to the 2 pixel step-up value and the fact that the horizontal component of her movement is applied first (see section 6.4.6). If the value obtained from this calculation were more than 2 pixels below the destination ledge, we would need to add more frames of umbrella and repeat the process. If we were to continue with the present value, we could instead remove some frames of umbrella and repeat, hoping to find a faster solution that still reaches the ledge.

When doing all of this, it is important to keep in mind that these calculations do not take Juni's initial subpixel position into account (it would be -100 in this case), nor do they consider subpixel anomalies (which can account for small inconsistencies whenever y-velocity changes from going up to going down or vice-versa). When creating a TAS, these calculations are a good starting point, but they are no substitution for actual testing.

It is also important to keep in mind that although the principle of even distribution allows for one-frame components, we cannot realistically retract umbrella and then deploy it again on the very next frame. It is possible for the very last umbrella dip to be one frame (because it is terminated by landing on the ledge, not opening umbrella), but every other interval must be either 0 frames or 2+ frames long. When the principle of even distribution requires more than one 1-frame umbrella interval, then we must do some re-arranging in order to still satisfy the input restrictions. A couple such re-arranging examples are shown in Fig. 15.

This entire process had been encoded into a python script. See section C. It is also available online at `http://www.ksruns.com/resources/umbrella_pump_calculator.py`.

8.3 Accelerated Wall Climbing

The standard wall climbing speed (holding up arrow next to a wall) is 205 cpx/f. Outside of casual gameplay, there are much faster ways to climb walls that involve taking advantage of the high vertical speeds of jumping. Recall that wall jumps and ground jumps are considered the same, and both start with a velocity of roughly 5 px/f upwards.

If we perform a regular wall jump, we obtain a 300 cpx/f horizontal velocity away from the wall (immediately becoming 200 if we are holding the direction towards the wall). Assuming no external interference, it will take Juni a minimum of 5 frames to regrab the wall, during which time she will swing 3 pixels outwards (the x-velocities during those 5 frames are -200, -100, 0, 100, and 200 centipixels per frame for a right wall). She is able to jump again on the first frame of reconnecting with the wall, which leads to 5 frame cycles of jumping.

If Juni has double jump, she is able to add double jumps into this 5 frame cycle, making it alternating cycles of 2 and 3 frames. It does not matter whether the split is 2 then 3 or 3 then 2 – that is simply a permutation of the same y-velocities over those five frames. This is the fastest glitchless way of wall climbing. However, there is a faster method still.

Recall from section 7.1 that facing away from a wall on the same frame as performing a wall jump will cause the 300 cpx/f horizontal boost to be applied backwards. By doing this, Juni is sent towards the wall, where she will collide and have her x-velocity return to 0. It is therefore possible to turn around on the very next frame and perform another wall jump. This technique of alternating glitchy wall jumps and facing the wall results in 2 frame cycles, which is the fastest possible wall climbing method.

Fig. 16 shows a comparison between the average speeds gained by the four different wall climbing methods. Note that accelerated wall climbing is the same with or without the double jump powerup. Another advantage of accelerated wall climbing is that it allows for more choice in when to leave the wall, without sacrificing too many frames. Note, however, that a standard jump may still be useful when jumping away from the wall or when rounding the top corner of the wall.

Ledge Fall Off	**Jump**	**Umbrella Pumps**	
⋮	⋮		
69	75		
92	98		
115	121	126	126
138			

(a) 36 frames, 3 umbrella pumps

Ledge Fall Off	**Jump**	**Umbrella Pumps**		
⋮	⋮			
69	75			
92	98			
115	121	126	126	126
		149		

(b) 37 frames, 3 umbrella pumps

Ledge Fall Off	**Jump**	**Umbrella Pumps**		
⋮	⋮			
69	75			
92	98			
115	121	126	126	126
138		149		

(c) 38 frames, 3 umbrella pumps

Ledge Fall Off	**Jump**	**Umbrella Pumps**		
⋮	⋮			
69	75			
92	98			
115	121	126	126	126
138	144	149	149	

(d) 39 frames, 3 umbrella pumps

Figure 15: Rearranging combinations to avoid having multiple umbrella pumps with only one frame. The red indicates the frames that were removed, and the blue indicates the frames that were added. Note that the 138 and 144 frames can be removed from the ledge fall off and jump and turned into an extra umbrella pump (because $126 + 149 < 138 + 144$).

Climbing Method	**Y-Velocities (Repeating) (cpx/f)**	**Average Y-Velocity (cpx/f)**
Basic	205	205
Jumping	460, 420, 380, 340, 295	379
Jumping with Double Jump	460, 415, 460, 420, 375	426
Accelerated	460, 415	437.5

(a) Without High Jump

Climbing Method	**Y-Velocities (Repeating) (cpx/f)**	**Average Y-Velocity (cpx/f)**
Basic	205	205
Jumping	477, 454, 431, 408, 363	426.6
Jumping with Double Jump	477, 432, 477, 454, 409	449.8
Accelerated	477, 432	454.5

(b) With High Jump

Figure 16: Comparison of the average speeds of various wall climbing methods.

8.4 Squeezing Past Shifts

When a Shift object is made invisible, it still retains the hitbox of the visible version. New level creators (and even some veteran ones) often don't realize that they need to change the Shift type as well in order to fully block off areas. This section will examine these kind of malconfigured Shifts and when it is possible to bypass them.

The default Shift hitbox is the glowing purple animation on the ground (see Fig. 7). It is a rectangle of 12 pixels wide by 6 pixels high, missing every alternate pixel on the top row. Because Juni is 17 pixels tall, she can fit above a Shift in a one-tile (24px) corridor with one pixel to spare. Note, however, there is one frame during the animation wherein the hitbox is 2 pixels taller. During this frame Juni cannot fit between it and a ceiling one tile above. This is important to keep in mind when attempting to jump over Shifts in confined spaces. The Shift animation will always start on the same frame upon loading a screen, however, so cycles will be consistent from screen transitions.

For the following calculations, we assume that Juni possesses the Run powerup. Juni's hitbox is 11 pixels wide, and the Shift hitbox is 12 pixels wide (11 if we allow for the missing pixel on one of the two corners). This means that there are 21 pixels where Juni and the Shift can come into contact. If Juni is moving at the maximum 3.5 px/f, then she will be in contact with the Shift for a minimum of 6 frames.

Combining all the above information, any jumps which desire to pass between a malconfigured Shift and a tile-aligned ceiling must be able to stay in a 2-pixel range of y-positions for 6 frames. Moreover, any jump which can stay within 0 or 1 pixels of a ceiling for 6 consecutive frames is capable of jumping over a malconfigured Shift on that ceiling.

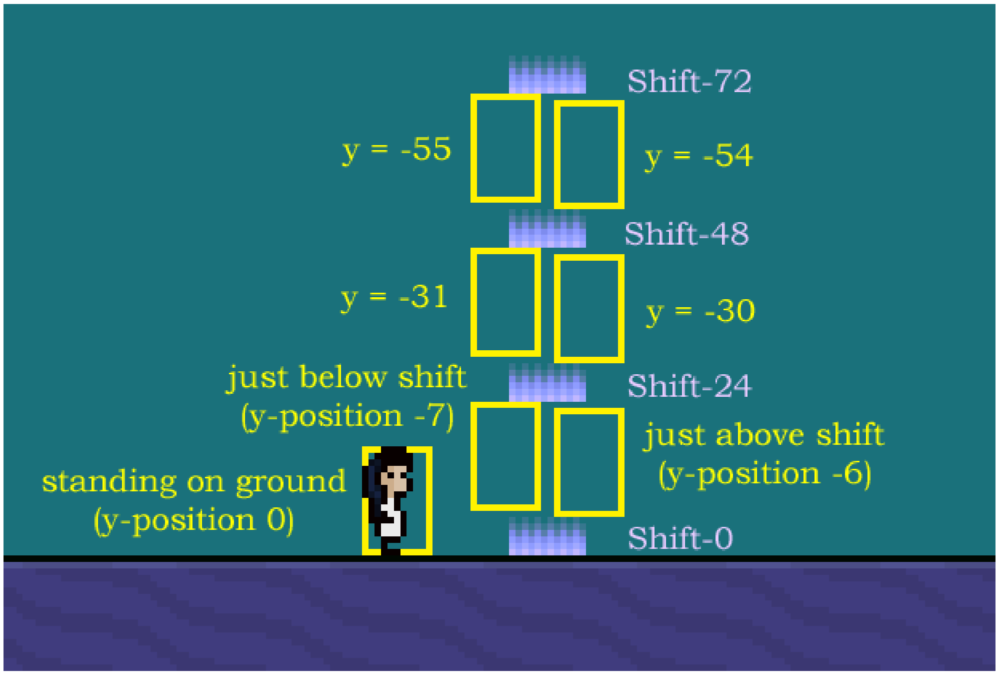


Figure 17: The relative coordinate system used for section 8.4

The rest of this section will be focused on determining which malconfigured Shifts are capable of being bypassed, and which are not. We shall not examine the feasibility of doing so, nor shall we look for the fastest methods. The purpose of this section is simply to show which are possible and prove which are not.

For the purposes of this section it will be useful to establish a standard coordinates system. Let us say that Juni's y-position when she is standing on the ground is 0, and (to stay in accordance with the screen position conventions) negative y-positions will indicate that Juni is above the ground. If there is a malconfigured Shift at ground level, then Juni will need to rise at least 6 pixels to jump over it (y-position -6 or higher). We shall call this Shift (one that is placed on the first non-ground tile) Shift-0, referring to the way in which it is 0 pixels off the ground. A Shift one tile above that shall be called Shift-24, two tiles above the ground Shift-48, and so on. Juni will come in contact with Shift-24 from y-position -30 (as with passing over Shift-0 at y-position -6) to y-position -8 (since she is 17 pixels tall). Thus, Juni will need to stay on y-positions -6 and -7 to squeeze between Shift-0 and Shift-24. She will need to stay at y-positions -30 and -31 to pass between Shift-24 and Shift-48, and so on. See Fig. 17.

We need to stay within 2 pixels vertically for 6 frames. It should be obvious that this can only occur at the peak of a jump, but here is a short proof by contradiction: Without loss of generality, suppose that Juni is travelling downwards. At minimum her y-velocity will be increasing by 23 cpx/f each frame. Assuming the first frame is not at the peak of her jump, then her starting y-velocity must be at least 1 cpx/f. The sum of her velocities over 5 frames must then be at least 235, which will cause her to move at least 2 pixels.

First, let us note that double jump is extremely powerful in bypassing malconfigured Shifts, because it allows us to control exactly what height we initiate a jump, as well as being able to double jump to stay close up against ceilings for longer. This essentially makes it possible to bypass all of the standard types of malconfigured Shifts. If there is a Shift-24, then we will be able to jump between it and either a Shift or a ceiling on the tile above (Fig. 18). If there is a ceiling 24 pixels above ground, then we will be able to use it to double jump above a Shift-0 (Fig. 19). If there is neither a Shift nor a ceiling on that tile, then we can simply jump through it.

Suppose now that we posses the High Jump powerup, but not the Double Jump powerup. Juni is still able to jump between a Shift-24 and either a Shift-48 or a ceiling, by letting go of the jump key to control her height, and then regrabbing it midair to slow her fall (Fig. 20). If the Shift-24 is replaced with nothing, we can obviously still jump through. The remaining case to examine is therefore what happens when there is a ceiling one tile above the ground. Although it is just barely possible to jump over a malconfigured Shift in such a one-tile-corridor if we have umbrella (Fig. 21), such a jump is not possible with only High Jump. We shall prove this fact rigorously:

Lemma 8.3. *Given a malconfigured Shift-0 (a Shift on the ground) and a ceiling one tile above that, it is not possible to jump from the ground and between the Shift and the ceiling without either the Double Jump or Umbrella powerups.*

Proof. First note that we are guaranteed to collide with such a low ceiling. With an initial speed of at least 460 upwards (440 with umbrella) and a maximum effective gravity of 45, there is simply no way to slow down fast enough to avoid it. Once Juni collides with the ceiling, her y-position is immediately set to -7, and her subpixel position to -100. The events before the collision therefore have no bearing on those after, and we may evaluate the two separately.

After the collision we desire to stay within the y-positions -7 and -6 for as long as possible. Juni starts at the higher of the two pixels, with a subpixel position of -100. This means that she can travel 300 cpx (putting her at y-position -6 with a maximum subpixel position of 100) before falling out of the safe zone. Without high jump, the lowest obtainable effective gravity is 40. Then Juni's subsequent y-velocities will be 40, 80, 120, and 160, and she will be able to stay within the two pixel margin for four frames (including that of the collision). With high jump, the best effective gravity we can manage is 23, with subsequent velocities of 23, 46, 69, 92, and 115, allowing her to maintain for one frame longer at five frames. Note that umbrella will change two of these velocities (160 to 120, and 115 to 103), but not sufficiently to add any more frames.

Because Juni needs to stay in the two-pixel range for 6 frames, we require additional frames before collision with the ceiling. Based on the number of frames that are available after collision, we require one frame with high jump, and two frames without. It is easy to show that the two frame without high jump are not possible. Our speed after breaching y-position -6 will be much greater 200 cpx/f, meaning that next frame will collide with the ceiling. With high jump, it is actually possible to have 1 frame in the two pixel margin, by taking advantage of the dampening properties of the umbrella (see Fig. 21). Without umbrella, however, it is not possible. Even with a maximum subpixel position pre-jump (100), the -477 cpx on the first frame is not sufficient to reach -6 (we get -5.77). Conversely, even with a minimal subpixel position pre-jump (-100) and a one frame jump for velocities of -477 and -432, we end up just barely entering the ceiling on the second frame (-8.09).

Without high jump we can obtain 1 frame above the malconfigured Shift before colliding with the ceiling and 4 frames afterwards, which is not sufficient to cross even with umbrella. With high jump we can obtain 1 frame before collision, though only with umbrella, and 5 frames afterwards, making the jump only possible with umbrella as well. (Note: The umbrella does cost us a little horizontal speed, but we have sufficient wiggle room.) Outside of double jump, none of the other powerups have an impact on Juni's y movement while jumping. Thus, without double jump or umbrella, this jump is impossible. □

Now that we have examined both the High Jump and Double Jump powerups, let us look at what jumps are possible with only Run and possibly Umbrella. We have already proven that jumps over a Shift-0 in a one-tile corridor are decidedly not possible. But if we replace the ceiling with a Shift, it simply become impossible to jump over it (the maximum jump height without powerups is only 29 pixels, whereas we require 30 to make it over a Shift-24. It is still possible to pass between two malconfigured Shifts with only the Run powerup, but we require a good y-position to jump from. This can either mean a diagonal slope nearby (or other non-tile aligned geometry) or that we posses the Climb powerup and there is a good wall nearby from which to jump. See Fig. 22 for an example of such a jump, using a starting subpixel position of 0. Note that jumps such as this, when done from a wall, can make it difficult to keep track of the subpixel position, which will obviously cause variations in the jump inputs required to make it work.

It is finally worth noting that it is possible to jump over a malconfigured Shift without any powerups (including Run) if it is placed directly against the top of the screen and provided a good place from which to jump. A summary of all this information may be found in Table 15.

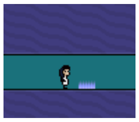
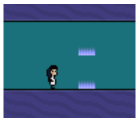
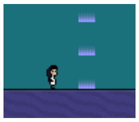
	🔴	🔴🟢	🔴🟢☂	🔴🔵	🔴🔵🟢
	NO Lem. 8.3		YES Fig. 21	YES Fig. 19a	YES Fig. 19b
 	MAYBE Fig. 22	YES Fig. 20		YES Fig. 18a	YES Fig. 18b

Table 15: Whether or not certain powerup configurations are able to bypass certain malconfigured Shift arrangements.

Input	Vel	Pos	Sub
	0	0	-100
S	-460	-3	60
S	-420	-7	80
S	-380	-11	60
⋮	⋮	⋮	⋮
S	260	-17	60
S	300	-14	60
	345	-10	5
	390	-7	95
	435	-2	30
S	-460	-6	90
S	-420	-11	10
S	-380	-14	90
⋮	⋮	⋮	⋮
S	-180	-27	90
S	-140	-29	30
S	-100	-30	30
S	-60	-30	90
S	-20	-31	10
S	20	-31	30
S	60	-31	90
S	100	-30	90

(a) Without High Jump

Input	Vel	Pos	Sub
	0	0	-100
S	-477	-3	77
S	-454	-8	31
	-409	-12	40
	-364	-16	4
	-319	-19	23
⋮	⋮	⋮	⋮
	401	-8	85
	446	-3	31
S	-477	-8	8
	-432	-12	40
	-387	-16	27
	-342	-19	69
⋮	⋮	⋮	⋮
	-207	-27	25
	-162	-28	87
	-117	-30	4
	-72	-30	76
	-27	-31	3
	18	-31	21
	63	-31	84
	108	-30	92

(b) With High Jump

Figure 18: Double jumping between a Shift-24 and either a ceiling or Shift-48

Input	Vel	Pos	Sub
	0	0	-100
S	-460	-3	60
S	-420	-7	80
S	0	-7	-100
S	40	-7	-60
S	80	-7	20
	125	-6	45
S	0	-7	-100
S	40	-7	-60
S	80	-7	20
S	120	-6	40

(a) Without High Jump

Input	Vel	Pos	Sub
	0	0	-100
S	-477	-3	77
S	0	-7	-100
S	23	-7	-77
S	46	-7	-31
S	69	-7	38
	114	-6	52
S	0	-7	-100
S	23	-7	-77
S	46	-7	-31
S	69	-7	38
S	92	-6	30

(b) With High Jump

Figure 19: Double jumping between a Shift-0 and a ceiling

Input	Vel	Pos	Sub
	0	0	-100
S	-477	-3	77
	-432	-8	9
	-387	-11	96
	-342	-15	38
	-297	-18	35
	-252	-20	87
	-207	-22	94
S	-184	-24	78
S	-161	-26	39
S	-138	-27	77
S	-115	-28	92
S	-92	-29	84
S	-69	-30	53
S	-46	-30	99
S	-23	-31	22
S	0	-31	22
S	23	-31	45
S	46	-31	91
S	69	-30	60

Figure 20: High jumping between a Shift-24 and either a ceiling or Shift-48

Input	Vel	Pos	Sub
☂	0	0	-100
S ☂	-477	-3	77
S ☂	-417	-7	94
S	0	-7	-100
S	23	-7	-77
S	46	-7	-31
S	69	-7	38
S	92	-6	30

Figure 21: Using Umbrella and High Jump to jump between a Shift-0 and a ceiling

Input	**Vel**	**Pos**	**Sub**
	0	18	-100
S	-460	14	60
S	-415	10	75
S	-370	6	45
S	-325	3	70
S	-280	0	50
S	-235	-2	85
S	-190	-4	75
S	-145	-6	20
S	-100	-7	20
S	-55	-7	75
S	-10	-7	85
S	35	-6	20
S	80	-6	100

Figure 22: Jumping between a Shift-0 and either a ceiling or Shift-24, using a starting position 18 pixels below ground level

9 Making a TAS

9.1 Routing

The first and most important step in any TAS is developing a route. Oftentimes levels may turn out to not be as clear cut as they appear in a casual playthrough. As such, it is usually best to start by looking for potential OOB locations. There are many different ways for OOB to occur, but the most common are around Shifts. If there are Shifts in the level, see if any can be exploited. Any Shifts that change Juni's screen position have a good chance of being used to get into a wallswim. If there are any Shifts that block Juni's progress, it is worth checking whether or not they are malconfigured.

The same goes for Warps, but to a lesser extent. If there are two different Warps on a screen, check if they can be abused. Remember that Warp behaviors in KS and KS+ are slightly different. If there is an area with a Warp on one screen but not another it may be possible to maneuver around it in a way the author didn't intend. For instance, there are Warps on screen x1086y1017 of *Saving Thalanill* that separate it from the rest of the map, but it is possible to fall down from x1086y1015 to x1086y1016, warp right to the disconnected screen, then immediately double jump back up into a wallswim.

Finally, it is worth checking for jumps that the author simply did not realize were possible. These jumps can be the cause for many out of bounds, especially with the use of some advanced techniques and glitches. Glitchy corner jumps and screen transition boosting are especially not well known by level creators.

Once all the useful exploits have been established, it is time to look at the victory requirements. In particular, look at the out of bounds possibilities. Because Juni can travel along the tops, sides, and bottoms of screens while OOB, a single OOB can make it possible to reach screens quite far away. If any of the out of bounds can drop Juni off at the end, or in an area that's close, they are worth considering. It may not be necessary to solve the problem of reaching the end as fast as possible, but simply reaching the OOB as fast as possible.

Once a basic end goal has been established (whether it be getting to the end or getting to a specific OOB), then it is time to begin routing. This is generally easier for linear levels and harder for open world levels. In general it is worth dividing the route into chunks. Find the bottlenecks in the routing (for instance, it is impossible to pass X area without first collecting everything that comes before it). This makes it possible to route small sections of the run at a time, rather than the whole thing. If the map is indeed open world (or the existence of many OOB make it basically open world), then there are often not many such bottlenecks. If that is the case, then consider dividing the map itself into different areas, and time how long it takes to complete each individual area (these can generally be rough timings). Then the timings can be fit together like puzzle pieces until most of the options are shown to be too slow and the fastest route emerges. If all else fails, the map can be further broken down to individual items and timings between them, and then a strong pathfinding algorithm applied to that to weed out the times. Even in massive open world levels, each powerup generally only appears a couple times. The number of nodes is therefore limited to roughly 30 at worst, which is solvable by modern algorithms.

9.2 Optimizing Movement

Once the route has been chosen, it is fairly clear what needs to be done on any given screen. This section is designed to provide some basic strategies for the majority of the TAS movement.

When moving through a screen horizontally, the goal is generally to hold a direction key and avoid any collision with walls. As long as Juni maintains a speed of ±350 cpx/f (or ±180 without Run), then it is guaranteed to be optimal. If this is not possible (Juni cannot make it around an obstacle fast enough), then usually the best attempt is to slow down as little as possible. Use knowledge of the movement engine to calculate exactly how many frames it will take to get enough height to go over or under the obstacle. Then adjust horizontal movement so that Juni is exactly at the edge of the obstacle (with maximum subpixel position) on the last frame before going around it. It is important here to keep in mind that horizontal movement is applied before vertical movement, and that she can cut a ledge short by 2 pixels because of the step-up.

When moving through a screen vertically downward, the goal is similar. Obtain terminal velocity as soon as possible, and try to maintain throughout the fall. If not possible, then cut the offending obstacle as close as possible. Sometimes this may mean landing on it and running off, sometimes it may mean performing an umbrella pump, and sometimes it may mean double jumping in midair, depending on the layout of the screen. It is worth noting that double jumps right against the ceiling of a block can delay a fall without costing as much as a double jump that doesn't hit a ceiling. In screens where a clean fall is possible, it is worth looking around for any platforms which may be edgebugged, because that may be able to give Juni an additional 1-5 pixels downwards.

When moving through a screen vertically upward with the climb powerup, the goal is generally to stay on the walls as much as possible, performing accelerated wall climbs. The one exception to this is when wind (15:5) objects are present. When it is required to jump between walls in order to keep climbing, then the goal should be to minimize the time between climbs. Keep in mind that the principle of even distribution states that it will always be better to split up this air time evenly, rather than have some short and some long (assuming the options have equal frames of airtime).

When moving through a screen vertically without the climb powerup, Juni will likely be jumping between platforms. Again, the principle of even distribution comes in quite handy. If there is a choice of intermediate platforms, it is always worth choosing the ones that are less evenly distributed. The more even the distribution, the more time Juni will spend covering the same vertical distance[20]. The same applies when performing a double jump in midair. The double jump should be the lowest one possible that still allows us to reach the platform.

Transitioning from horizontal movement to vertical movement is fairly straightforward. If we are transitioning to a wall climb, we want to be as high on the wall as possible when we start. If given liberty to jump from anywhere, then we can precisely manipulate the jump(s) in order to be as high as possible with the highest possible subpixel position when we touch the wall. If we are forced to jump sooner or later than ideal, then apply the principle of even

[20] Note that when we talk about even distribution for umbrella pumps it is the opposite: we want Juni to be covering less vertical distance, so that she stays higher up at the end.

distribution – keep the double jump evenly spaced. If we have the umbrella things become a bit more complicated, and it make take some math or programming to calculate the optimal balance between using umbrella to maintain height and freefalling to hit the wall sooner.

When transitioning from horizontal movement to a fall, it is often worth jumping beforehand to begin building up vertical speed downwards. The subpixel position manipulation is less necessary in this scenario, because an edgebug will generally snap Juni to the correct pixel anyway. If Juni is travelling at terminal velocity, the edgebug is all that is required to guarantee optimality. If not, then it may be worth doing a little manipulation just to make sure that her fall speed before the edgebug is maximal.

Finally, it is worth considering the fastest way to jump out and around an overhang, then turn back and grab that wall. The first step here is usually to calculate the first frame that Juni will clear the corner, and then plan a jump accordingly. If she will be a couple pixels beyond the wall by that point it is safe to begin turning around earlier to just barely miss the corner. Once that has been determined, calculate what frame Juni will regrab the wall. From there it is a simple matter of seeing how many frames Juni will spend airborne between the ground and the wall, and planning the double jump accordingly to gain as much height as possible.

9.3 Conclusion

This section covers the general theory and methodology of making a TAS. No doubt actual TASers will encounter situations that require much more calculation and planning than the ones listed here, but those sections will also likely be the most fun to watch. Sometimes there may even be multiple paths through a screen or through an area that are simply too close to eyeball, and both of them must be timed out. This guide is meant to be comprehensive in analysis of the movement engine, but to be comprehensive in the types of screens that may be encountered is nigh impossible. It is therefore that I conclude this handbook here. I believe that anyone who sincerely wished to do any kind of TASing will now have the tools to do so, and I trust that they will have the passion to find their own way through whatever challenges that level may present. Therefore good luck, and enjoy the ride.

A Enemies

A.1 Preliminaries

The movement and firing patterns of enemies are not handled in the main Event List, but in their own Object Behaviors. These object behaviors are universal for all objects of a given type. This means that all enemies of the same type that are on screen together will share the same behavior code. This will often cause actions such as changing direction and firing projectiles to occur simultaneously for all enemies of the same type, unless each individual enemy is specifically given its own timing variables.

All of the Object Behaviors contain a "Soul" group, which contains most of the code. The one line of code not within that group will tell the game to only activate the Soul code when there is at least one enemy of that type currently loaded. This means that Juni being in a screen with none of that enemy type will cause the enemy behavior to freeze.

Many of the objects make use of Every events, which are a built-in component of MMF2. Every events keep a global timer running, which can be used to, for example, fire a bullet Every 03"-50. The notation 03"-50 refers to 3 seconds and 50 centiseconds, or every 3.5 seconds. Recall from section 5.4 that Every events are relative to the system clock, not the frame count. This means that anything being triggered by an Every event is not guaranteed to always happen on the same frame, depending on framerate fluctuations, and may happen far more often than normal when experiencing extreme slowdown.

Additionally, these Every events will only trigger when their corresponding code is active. If the Soul for a specific enemy type is deactivated, then the countdown to the next firing of an Every event will pause. When Juni re-enters a screen with that enemy, the Soul will re-activate, and the countdown will continue. Note that because the Object Behaviors are universal, it does not have to be the same screen. All enemies of the same type will obey the same global Every timers. When the game is first loaded, all of the Every event countdowns will be treated as if they were just activated. This often means that the longest time before an enemy performs some action (moving, shooting, etc.) can be seen when first encountering it. From there on, all enemies of that type will perform that action on a fixed timer, pausing that timer when Juni leaves the screen, and resuming when she enters another screen with the same enemy.

There is one more type of timer that is implemented on some enemies. Unlike the built-in MMF2 Every events, these timers have been created by Nifflas, and use a countdown variable tied to the enemy object. In contrast to behaviors, variables are specific to the enemy, and are reset every time a new enemy is created (and discarded when that enemy is destroyed). This means that enemies using this kind of timer can become desynchronized. Even without desync, this kind of timer has an interesting way of causing MMF2's "Pick one" events to fail. Because all the enemies reach 0 on the counter simultaneously, the "Pick one" event will fire once for every enemy on screen, effectively picking all of them. This means that enemies attempting to pick only one to perform an action will fail when using this kind of timer (a good example is 2:6 Nasty Flything, which all shoot at once). It also means that their initial state will be identical every time loading a screen with them, as opposed to the Every events which remember their cycles between screens. The notation we shall use to denote these kinds of timers will be "Frames X (init Y)."

Speaking of notation, some of the terms we will be using are explained in Fig. 23. In summary, these are the important things to keep in mind:

Every Events

- Cycles are the same for all enemies (of the same type).
- Cycles only update when at least one enemy is loaded.
- Can last fewer frames if the game experiences slowdown.
- Can be manipulated for speedrun purposes.

Frame Countdowns

- Cycles are unique for each enemy.
- Cycles are reset on every new screen.
- Always last the same number of frames.
- Don’t work with “Pick one” events.

Every SS”-CC	An action occurs every SS seconds and CC centiseconds, using MMF2’s “Every events”.
Frames X (init Y)	Using a Nifflas-created counter, with starting value Y and decrementing by 1 every frame (including the initialization frame). If it hits zero, an action is triggered and the value goes back up to X.
Always	An action triggers every frame.
Never	Not strictly part of the code, but this is used to indicate that an enemy doesn’t perform an action which many others do.
Random(N)	A randomly chosen integer between 0 and N-1 (inclusive).

Figure 23: Important Notation

A.2 Bouncing Bullet, and Teleporting Rabbits

Some objects that use the Bouncing Ball movement (section 5.3) also belong to the qualifier class Bullets. These “Bouncing Bullet” objects are confined to precisely two directions, left and right, and they have additional code written by Nifflas that tells them to turn around when colliding with a wall. The way this code is programmed can lead to some interesting effects.

The most basic turn-around code is when a Bouncing Bullet leaves the screen on the left or right. If its x-position is less than 0 (leaving the left), the game will tell it to face to the right and set its flag 31 on. Flag 31 is the designated "just hit a wall" flag. Because the bouncing bullet behaviors are all batched together, this is used to tell individual objects (like the 4:9 Spider) when they have hit a wall. Similarly, if the object leaves the right side of the screen (x-position is more than 600), it will face to the left and set flag 31 on.

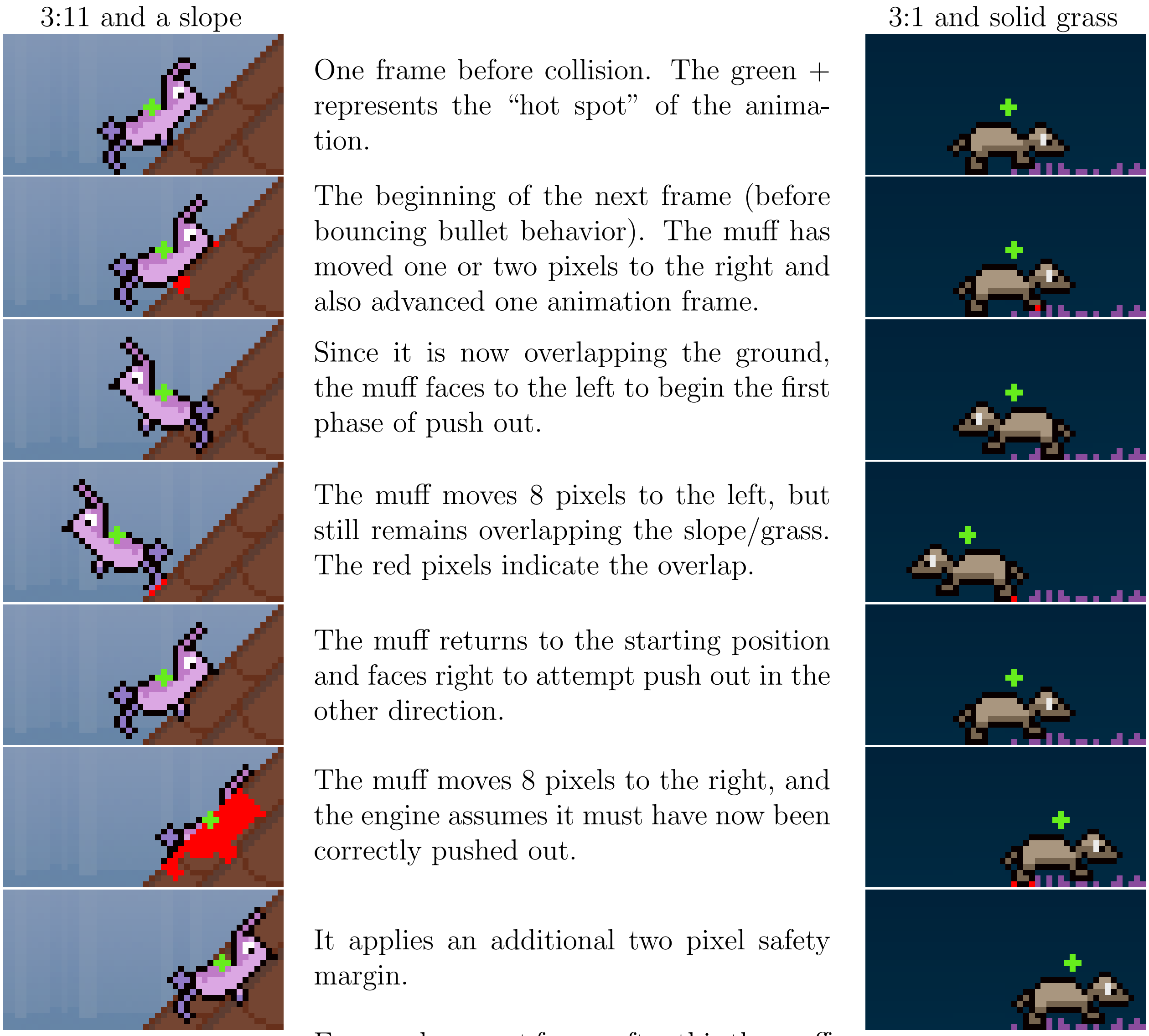
If a Bouncing Bullet object collides with a wall, it activates a more complex Push Out script. First the game attempts to eject it from the wall to the left, one pixel at a time. If the object successfully leaves the wall, then it will not push it any farther. If, however, it is still overlapping the wall after moving 8 pixels to the left, then the object will return to the starting position and attempt the same push out to the right. This time, if the object has not left the wall after 8 attempts, it will not return to its starting position. Rather, the game assumed it must have left the wall by then. After the object is presumed to have been safely ejected, the game applies an additional 2 pixel safety margin in whatever direction it is facing (i.e. move it 2 pixels left if facing left, and 2 pixels right if facing right). Finally, flag 31 is set on.

Under ordinary circumstances, this code works fine. However, it can become glitchy when low-height obstacles are combined with animations that involve lifting off the ground. Making solid grass, rocks, or other decorations is a common offender, but for this example we shall be looking at diagonal slopes and the 3:11 Fast Rabbit. The rabbit begins by hopping towards the slope. Recall that, unlike Juni, all other object hitboxes are based on their animation. Thus, the rabbit has not yet triggered a collision, even though its center of gravity is well over the ground. The next frame the rabbit does collide with the wall. The game attempts to push it back out to the left. During this process, it also tells the rabbit to face to the left. This causes the hitbox to change as well, causing the push out to fail. The game therefore switches to attempting to push it out to the right. This is also not possible, but after 8 pixels the game must assume that it was. The rabbit is now 8 pixels deep in the ground. See Fig. 24.

Whenever a bouncing bullet object ends up inside of the ground like this, it will attempt to perform a push out every frame. Each push out will move the object to the right 8 pixels, plus another 2 pixels from the safety margin. This causes the object to move towards the right at 10 px/f, in addition to whatever speed it may have from its normal movement. This rapid rightward movement will continue until the object either leaves the wall on the other side, or is pushed all the way to the right edge of the screen, where it becomes trapped. This is what causes teleporting rabbits – when a rabbit clips into the ground in this fashion and appears to teleport very quickly to the right.

Performing too many push out cycles for bouncing bullets tends to make MMF2 glitch out[21]. It is unclear exactly what the code failure is, but the objects seem to enter a state much like atoms in a Bose–Einstein condensate. They lose their identity and start to share information and behaviors, such as whether or not they overlap a wall, how they should respond to the push out code, sometimes even appearing to be moving both left and right

[21] There is a good chance this is an intended feature of MMF2. It seems as though it detects when too many calculations are going on at once, and then decides to take action to make it less CPU intensive. That action seems to be sharing the results of collision detection calculations between multiple objects or merging fast loops for multiple objects into one. Or perhaps it simply runs off the end of a calculation stack and can only use the last result. Regardless, it improves performance at the cost of glitchiness in certain rare circumstances.



3:11 and a slope

3:1 and solid grass

One frame before collision. The green + represents the "hot spot" of the animation.

The beginning of the next frame (before bouncing bullet behavior). The muff has moved one or two pixels to the right and also advanced one animation frame.

Since it is now overlapping the ground, the muff faces to the left to begin the first phase of push out.

The muff moves 8 pixels to the left, but still remains overlapping the slope/grass. The red pixels indicate the overlap.

The muff returns to the starting position and faces right to attempt push out in the other direction.

The muff moves 8 pixels to the right, and the engine assumes it must have now been correctly pushed out.

It applies an additional two pixel safety margin.

Every subsequent frame after this the muff will continue to be pushed 10 pixels to the right until it comes out the other side.

Figure 24: Visual representation of the steps that occur during a Push Out cycle that causes Teleporting Rabbits.

at the same time. Eventually this results in every successful push out persisting indefinitely. That is, the objects will continue to add the +2 or -2 safety margin every single frame until it either collides with another wall or something else in the object behavior tells it to stop moving. This adds 2 px/f to the speed in whichever direction it is facing, creating the appearance of hyperspeed bouncing bullet movements.

A.3 Enemy Behaviors

A.3.1 1:x Multiple Liquids

There are eight types of liquid, 1:1, 1:2, 1:5, 1:10, 1:12, 1:14, 1:19, and 1:22. As with most enemies in the game, the liquids' hitboxes match their animations, meaning that they are constantly changing. Most of the liquids have fixed heights on the left and the right side of the tile, while the liquid in the center forms a curve that moves up and down. When the curve is at its lowest, it can make jumps possible lower than normal. When the curve is at its highest, it can prohibit jumps too close to the liquid surface. The only animation with a different behavior is that of 1:5 Lava. It maintains a core hitbox 12-pixels high (half way up the tile), which is then overlaid with small, medium, and large splashes. Although probably just for decoration, the largest splash can reach 4 pixels above the ordinary lava surface, and will still kill Juni on contact, making this one of the trickier liquid animations to deal with.

Additionally, the starting frame and speed of animation are all chosen randomly when the objects are spawned on screen load. This element of randomness makes it impossible to have any sort of consistent cycles. Most of the liquids choose an animation speed between 20 and 29 inclusive, with the exception of the two dark blue liquids 1:10 and 1:12, which use a speed between 30 and 44 inclusive.

A.3.2 1:x Underwater Blocks

The underwater blocks (1:7, 1:8, 1:9, 1:11, 1:17, 1:21, and 1:24) are different from the surface of the water in that they are not deadly. In fact, they behave exactly as any other solid wall tile.

A.3.3 2:x Flythings

All of the Flythings (2:1, 2:2, 2:5, 2:6, 2:7, 2:14, and 2:22) and the two Propeller Dudes (2:15 and 2:24) follow the same pattern of movement. They have two variables, TravelingX and TravelingY that indicate a rough direction of movement. When they are first loaded with a screen, these values are randomized to either -1, 0, or 1. These values are then re-randomized every 2-3 seconds or so. In addition, they are re-randomized if the enemy ends up overlapping a solid object or leaving the screen, in which case the enemy will also return to the last position it was not inside of collision or outside of the screen. This means that being spawned inside of a wall will cause them return to the previous position every frame, effectively not moving at all. Every frame, the Flythings will move by adding TravelingX and TravelingY to their current x and y positions respectively, plus an additional random value between -1 and 1 inclusive that causes the small vibrations as they move.

Quad Flything and Fall Propeller Dude both have a slight modification to this code. Unlike the other Flythings, neither of them will randomly change direction in midair. The only time they will change directions are when bumping into collision, leaving the screen, or when first loaded. Additionally, if the direction they choose when first loaded is TravelingX=0 and TravelingY=0, then they will re-randomize their direction one more time. This makes it significantly less likely for them to be holding still when first loaded than the other Flythings. It is also possible for them to hold still upon colliding with something, but the other randomness present in their movement makes it likely that they will collide with it again and begin moving once more.

On top of this, when Fall Propeller Dude randomizes TravelingY, it can only be 0 or -1. The only time TravelingY can be greater than 0 is when the difference in x-position between Believed Position and Fall Propeller dude is less than 55, and Believed Position is more than 24 pixels below Fall Propeller Dude. Then it will enter attack mode, where TravelingY will be 3, and TravelingX will be plus or minus 2, depending on which side Juni is on. If Juni has exactly the same x position, it will maintain whatever the previous value was. The only way to break out of attack mode is to once again collide with a wall or leave the screen.

Of the Flythings that can shoot, they will do so on an Every interval as well, with the exception of Nasty Flything. Nasty Flything has an internal timer that is initialized to 25 whenever the enemy is loaded. The timer will decrement by 1 every frame, and then check if it has reached 0. If so, it will fire four shots and then set the timer back to 124, whereupon the cycle repeats. Bad Flything and Another Bad Flything will spawn four or six shots. Tentacle Flything and Quad Flything will initiate countdowns, firing one round of shots per frame until the countdown hits 0. Tentacle Flything will fire 15 rounds of 2 shots each, and Quad Flything will fire 9 rounds of 4 shots each. If Juni leaves the screen during one of these countdowns and re-enters, then the flyer will not continue to fire. This means that the greatest "opening" we can obtain between shots is by loading them one frame after beginning to shoot in their cycles, thereby causing them to not shoot when they otherwise would be. Because of the nature of Every events, this happens the first time they are loaded at all, which makes it seem as though the first cycle is longer than all subsequent ones.

Tentacle Flything and Quad Flything are unlike the Bad Flythings in that they will "Pick one" during the shooting action, rather than deciding when to shoot. This means that any of them are capable of shooting an individual projectile, but the total number of projectiles created is fixed. When there are a large number of them on screen at once, each individual enemy's spray will become sparse.

The frequencies with which the Flythings will randomize their TravelingX and TravelingY, as well as the frequencies with which they will begin shooting are listed in table 16.

Flything	**Direction Change Frequency**	**Shooting Frequency**
2:1 Flything	Every 03"-00	
2:2 Bad Flything	Every 03"-00	Every 03"-24
2:5 Another Bad Flything	Every 02"-00	Every 02"-50
2:6 Nasty Flything	Every 02"-00	Frames 124 (init 25)
2:7 Flything	Every 03"-00	
2:14 Tentacle Flything	Every 03"-00	Every 02"-00
2:15 Propeller Dude	Every 03"-00	
2:22 Quad Flything	Never	Every 02"-25
2:24 Fall Propeller Dude	Never	

Table 16: Flything Values

A.3.4 2:x Bees

The bees obey a simple bouncing bullet movement, with the red bee moving slightly faster. The shots they fire are on an Every timer, with an additional element of randomness determining whether or not they will shoot.

Bee	Movement Speed	Shooting Frequency	Chance of Shot
2:9 Bee	7	Every 00"-45	1 in 4
2:10 Red Bee	9	Every 00"-25	1 in 5

Table 17: Bee Values

A.3.5 2:x Homing Shapes

The homing shapes all obey a bouncing ball movement with constant speed. Whenever the homing shapes are not overlapping the Believed Position, they will change direction towards it. When overlapping the Believed Position, they will continue on their current trajectory. This is what causes them to oscillate back and forth around a hologram, or upon a successful kill. For unknown reasons, the Fast Homing Square will aim 3 pixels higher on the Believed Position than the other two homing shapes. The Homing Circle and Homing Rectangle have two speeds, a slow one for when they overlap a wall, and a faster one for when they do not. They will also change animation when inside a wall, to a sequence which has a changing hitbox. Depending on the state of this animation, they may consider themselves no longer overlapping a wall at different times. The Fast Homing Square likewise has slow and fast speeds, but it will move slowly when overlapping Believed Position, and quickly when not. Because the Believed Position lags one frame behind Player Position, it is possible to move fairly slowly (most notably, walking speed) away from the Fast Homing Square and not be killed by it. It will nearly reach Juni, but slow down as soon as it touches the Believed Position, causing it to follow closely behind her but never able to catch up.

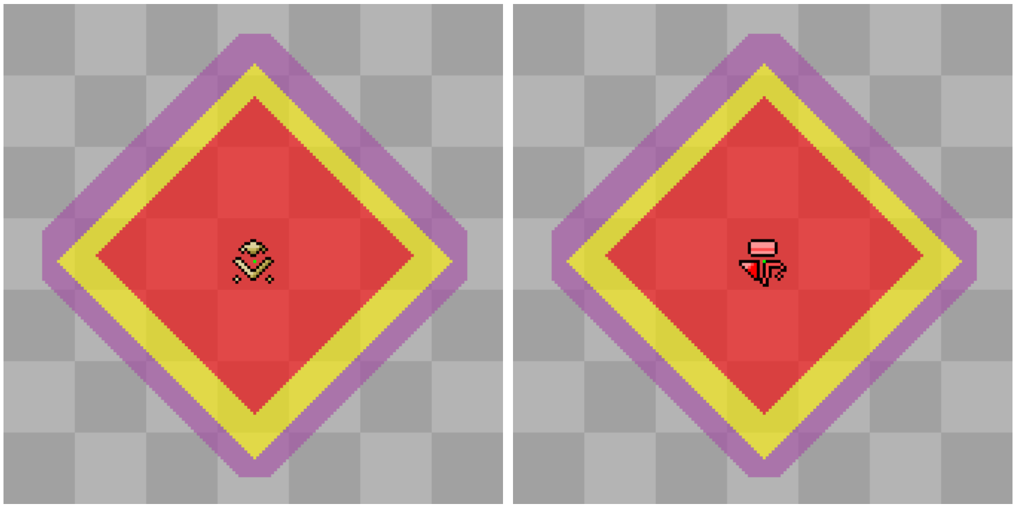
Homing Shape	Fast Movement Speed	Slow Movement Speed
2:11 Homing Circle	7	3
2:12 Homing Rectangle	10	4
2:32 Fast Homing Square	20	7

Table 18: Homing Shape Values

A.3.6 2:x Elementals

Every frame, the elementals use Juni's position to calculate a Distance to Player, using the taxicab distance (the distance between x-positions plus the distance between y-positions). When this distance drops below 67, the player is killed and the elemental plays a zapping animation. See Fig. 25 for a visualization of this hitbox. Note that these two events are distinct, so even if it is possible for Juni to avoid the hitbox of the animation, she will still be killed.

There is a safety mechanism that prevents the elementals from firing whenever Juni is closer than 18 pixels to the edge of the screen (that is, $x \leq 30$, $x \geq 569$, $y \leq 35$, or $y \geq 215$). This safety zone is designed specifically for the Player Position, whereas the attack detection uses Believed Position. Consider a situation in which Juni places a hologram inside the radius of the elemental while also inside the safety zone. Then the elemental will kill Juni as soon as she leaves the safety zone, even if she is far away from the elemental itself.



(a) 2:18 Lightning Elemental

(b) 2:19 Fire Elemental

Figure 25: Visualization of Elemental "hitboxes". The gray background is a tile-aligned grid. The hotspots of the elementals are marked with a green pixel. The yellow area indicates the pixels with taxicab distance less than 67 from the hotspots. When Juni's hotspot is in the yellow area, she will be in the attack radius, and when her hotspot is outside the yellow area, she will not. When Juni's Player Position is fully inside of the purple area, then she will be in the attack radius, and when it is not, she will not. When Juni is in contact with the red area, then she will be in the attack radius. Note, however, that the red area is not an exact hitbox, because Juni can be next to the corners without touching the red area but still activating an attack.

A.3.7 2:x Gap Jumpers

The gap jumpers have a constant XSpeed, which is added to their x-position every frame. When first loaded, this XSpeed is 4 for the right-facing one, and -4 for the left-facing one. After colliding with a backdrop while moving, the gap jumpers will turn around and push themselves out of the wall 1 pixel per frame (to observe this push out, place one inside of a wall). The collision will also trigger a Countdown To Flight which starts somewhere between 80 and 159 (inclusive), and decreases by one every frame (including the first). When this countdown hits 1, the XSpeed is set to either 3 or 4 (or -3 or -4 if facing left), and the gap jumper begins to move again. The Countdown To Flight does not wait for push out to finish, so they can turn around while still inside a wall if it is long enough.

Gap Jumper	Initial X Velocity	Countdown to Flight
2:20 Gap Jumper	4	80+Random(80)
2:21 Gap Jumper	-4	80+Random(80)

Table 19: Gap Jumper Values

A.3.8 3:x Mufs

The base movement for the Mufs is fairly basic Bouncing Bullet movement. Every so often they will change direction to face randomly left or right, and every so often they will change to a random speed (both using Every Events). They also start with a random left or right direction. There is not much of particular interest to discuss here, aside from the fact that the initial speed is not always in the set of potential random speeds, meaning that some Muff speeds are only attainable when first loading a screen.

Muff	Change Direction	Change Speed	Initial Speed	Random Speeds
3:1 Friendly Mufs	Every 04”-00	Every 00”-70	8	{0,0,5,10}
3:2 Roller Mufs	Never	Every 01”-10	8	{5,10}
3:3 Blue Mufs	Every 04”-00	Every 00”-77	2	{0,0,2,4}
3:4 Afraid Muff	Every 03”-00	Every 00”-76	8	{0,0,5,10}
3:6 Angry Muff	Every 02”-00	Every 00”-76	8	{0,10}
3:7 Flat Mufs	Every 04”-00	Every 01”-00	4	{0,0,3,6}
3:8 Another Roller Mufs	Never	Every 01”-10	8	{5,10}
3:9 Round Rabbit	Every 03”-67	Every 01”-12	0	{0,0,5,8}
3:10 Regular Rabbit	Every 04”-54	Every 00”-86	0	{0,0,5,10}
3:11 Fast Rabbit	Every 03”-00	Every 00”-65	0	{0,0,9,14}
3:15 Forest Girl	Every 04”-55	Every 00”-66	8	{0,0,0,4,5,6}
3:16 Hamster	Every 05”-00	Every 00”-80	8	{0,0,5,10}
3:17 Hamster 2	Every 03”-38	Every 00”-65	8	{0,0,5,10}
3:18 Hamster 3	Every 04”-26	Every 01”-00	8	{0,0,7,12}
3:19 Purple Lizard	Every 03”-98	Every 00”-71	8	{0,0,5,11}
3:20 Cute	Every 08”-00	Every 00”-70	0	{0,6}
3:22 Spiker	Every 02”-00	Every 00”-76	8	{0,10}
3:23 Walker	Every 01”-85	Every 00”-70	8	{0,10}
3:24 Hider	Every 02”-85	Every 00”-65	0	{0,8}
3:28 Forest Boy	Every 04”-05	Every 00”-72	8	{0,0,0,4,5,6}
3:30 Flying Person	Every 03”-97	Every 00”-66	8	{3,6,9,12}
3:31 Fly Boy	Every 04”-15	Every 00”-77	8	{0,0,0,4,5,6}
3:33 Another Afraid Muff	Every 03”-00	Every 00”-76	8	{0,0,5,10}
3:34 Mini Camel	Every 05”-55	Every 02”-05	4	{0,4}
3:35 Mini Camel Rider	Every 05”-35	Every 03”-25	3	{0,3}

3:36 Green Lizard	Every 03”-98	Every 00”-71	8	{0,0,5,11}
3:40 Cave Boy	Every 04”-00	Every 00”-70	8	{0,0,0,4,5,6}
3:44 Grey Lizard	Every 04”-98	Every 00”-82	8	{0,0,5,11}
4:17 Hedgehog	Every 02”-00	Every 00”-60	8	{0,0,5,10}
4:19 TriShot	Every 02”-00	Every 00”-76	8	{0,10}
4:20 FiveShot	Every 01”-90	Every 00”-84	8	{0,10}
5:2 Shadow Person	Every 04”-00	Every 02”-70	0	{0,4}
5:3 Shadow Person	Every 05”-12	Every 02”-36	3	{0,3}
12:3 Ghost Mufs	Every 04”-00	Every 00”-70	8	{0,0,5,10}

Table 20: Muff Values

3:5 Curious Muff and 3:25 Ratlike are not included in the above table, because they do not use a Muff movement. Rather, they will attempt to move 1 pixel per frame towards Juni when far enough away (greater than 16 pixel difference in x position) and on the same level (less than 12 pixel difference in y position). The rest of the objects in Bank 3 are all stationary. 12:3 Ghost Mufs was included in this list as well, because it doesn't obey any of the properties that the other ghosts share (travelling through walls or changing transparency).

On top of the Muff movement, a few enemies have additional behaviors. For the two Roller Mufs, one is chosen at random to attack Juni on an Every Event timer (Every 04”-00 for 3:2 Roller Mufs, and Every 06”-00 for 3:8 Another Roller Mufs). During their attack, the bouncing ball movement speed is set to 0 (so that it does not interfere), and a different variable, RollSpeed, is set to either 3 or -3. If Juni's x-position is to the left of the roller muff, the RollSpeed is set to -3. If Juni's x-position is equal to or greater than that of the roller muff, then RollSpeed is set to 3. This value is added to the roller muff's x-position every frame until the bouncing bullet code informs it of a collision, at which point it reverts to regular Muff movement.

The other enemy class to be encountered among the Mufs are the Spikers. Just as the Roller Mufs will never change direction, the spikers will change direction to either face towards or away from Juni (and if they have exactly the same x-position as Juni at the time, they will continue their current direction). Spikers have the same DistanceToPlayer variable as the Elementals, which they calculate using the same taxicab distance. If that distance drops below a certain threshold, the speed will be set to 0, and the spiker will deploy its spikes and become deadly. After becoming spiky, if the distance then rises above a second threshold, then the Spiker will play a Stand Up animation. After four frames of that unspiking animation, it will revert to normal Muff movement, set its speed to 10, and no longer be deadly. Because the stand up animation speed is 35 (35/100ths of animation frame per game frame), it will take roughly 12 frames to revert. The fact that it is deadly during that time is only really relevant when Shifts or same-screen Warps are involved. Otherwise it is not possible to leave the spiker's radius, place a hologram, and then cross the distance in time to reach the spiker before its spikes have fully retracted.

4:19 TriShot and 4:20 FiveShot behave similarly to the spikers, except that instead of becoming dangerous to touch when they attack, they instead shoot fireballs. When they sit down a shot countdown is initialized to 10. The countdown decrements by 1 every frame, shooting when it reaches 0. The counter is then reset to a random value between 60 and 79 inclusive, and continues to count down to 0. Because the initial countdown is so much faster than subsequent countdowns, they can be made to shoot much more frequently than normal by quickly exiting and re-entering the attack radius. The stand-up animation speeds for these two are both 50, meaning it takes them 8 frames to stand up.

Spiker	**Direction Change**	**Spike Distance**	**Stand Up Distance**
3:4 Afraid Muff	Away from Juni	< 90	> 120
3:6 Angry Muff	Towards Juni	< 60	> 100
3:22 Spiker	Towards Juni	< 40	> 70
3:23 Walker	Towards Juni		
3:24 Hider	Away from Juni		
3:33 Another Afraid Muff	Random	< 90	> 120
4:19 TriShot	Towards Juni	< 140	> 160
4:20 FiveShot	Towards Juni	< 140	> 160

Table 21: Spiker Values

A.3.9 4:x Crawlers

The crawlers have a simple walk-towards-Juni behavior. All of them will trigger when they are greater than 4 pixels away from Juni on their movement axis (that is, horizontal-moving crawlers will move towards Juni when the difference in x-position is greater than 4, and vertical-moving crawlers will use the difference in y-position). All of the crawlers move at 1 pixel per frame, except for 4:8 Slow Side Circle Crawler, which moves 1 pixel Every 00”-03, or 2 out of every 3 frames (assuming correct sync with the every event). If a crawler’s movement causes it to overlap a wall, then it will retract that move. This will also cause their walking animation to stop and set a variable called “wall touch” on.

4:6 Drop Crawler will attack when Juni’s x-position is closer than 200 pixels, and 4:7 Drop Crawler 2 will attack when Juni’s x position is closer than 100 pixels. The floor drop crawler attacks using an every event, and the ceiling drop crawler attacks with a frame timer, giving them different behaviors. The floor drop crawler has global cycles, and it’s possible to repeatedly get within the attack radius without triggering an attack. The ceiling drop crawler is always reset when loading a new screen. Its attack timer will always count down regardless of Juni’s position, meaning that it will generally shoot as soon as Juni enters its radius. It is also possible for multiple to be shooting at once, unlike the ground drop crawler.

In the following table X_{diff} stands for the distance between the x-positions of the Crawler and Believed Position (that is, $|X_{\text{Crawler}} - X_{\text{BP}}|$), and likewise for Y_{diff} with y-positions.

Crawler	**Move When**	**Move Speed**	**Attack When**	**Attack Rate**
4:1 Circle Crawler	$X_{\text{diff}} > 4$	1 px/f		
4:2 Side Circle Crawler	$Y_{\text{diff}} > 4$	1 px/f		

4:6 Drop Crawler	$X_{\text{diff}} > 4$	1 px/f	$X_{\text{diff}} < 200$	Every 01”-88
4:7 Drop Crawler 2	$X_{\text{diff}} > 4$	1 px/f	$X_{\text{diff}} < 100$	Internal Timer 80 (init 25)
4:8 Slow Side Circle Crawler	$Y_{\text{diff}} > 4$	1 px Every 00”-03		
4:13 Side Circle Crawler	$Y_{\text{diff}} > 4$	1 px/f		

Table 22: Crawler Values

The drop crawlers both shoot out a maximum three fireballs at a time, one to the left, one towards the middle, and one to the right. If the “wall touch” variable is on (that is, they are standing next to a wall), then they will only spawn two fireballs each, and forego the one in the direction of the wall. Each fireball (which use a bouncing ball movement) has three directions in which it may be shot, and a myriad of speeds. The ceiling drop crawler shoots its fireballs in a more spread out pattern than the floor drop crawler, presumably because the floor drop crawler’s shots stay on screen longer, thereby allowing them to spread out more naturally.

Drop Crawler	**Shot Type**	**Shot Direction**	**Shot Speed**
4:6 Drop Crawler	Right side	3, 4, or 5	25 + Random(15)
	Middle	7, 8, or 9	45 + Random(20)
	Left side	11, 12, or 13	25 + Random(15)
4:7 Drop Crawler 2	Left side	17, 18, or 19	25 + Random(15)
	Middle	23, 24, or 25	25 + Random(10)
	Right side	29, 30, or 31	25 + Random(15)

Table 23: Drop Crawler Values

A.3.10 4:x 6Legs

The 6Leg family all obey the same basic movement. When Juni is more than 4 pixels away horizontally and within a certain range vertically, then they will move horizontally towards her. If that horizontal movement causes one to end up in a wall, then it will return to its position before that move.

A.3.11 4:x Strangers

The strangers all obey virtually the same movement. They all have a Y Speed variable which takes the values -1, 0, and 1, and is initially set to -1. Every frame, this Y Speed is added to the y-position. If that causes the stranger to enter a wall, then it will back out by moving twice as far in the opposite direction. It will also change the sign on Y Speed if that happens. The strangers all shoot at intervals determined by an Every Event. During the shooting animation (which lasts 10 frames) Y Speed is set to 0. On frame 5 of the shooting

6Leg	Move When	Move Speed
4:3 6Leg	$Y_{\text{6Leg}} < Y_{\text{Believed Position}} + 36$ $Y_{\text{6Leg}} > Y_{\text{Believed Position}} - 12$ $X_{\text{diff}} > 4$	3 px/f
4:4 6Leg Child	$Y_{\text{6Leg}} < Y_{\text{Believed Position}} + 36$ $Y_{\text{6Leg}} > Y_{\text{Believed Position}} - 12$ $X_{\text{diff}} > 4$	2 px/f
4:5 6Leg Baby	$Y_{\text{6Leg}} < Y_{\text{Believed Position}} + 12$ $Y_{\text{6Leg}} > Y_{\text{Believed Position}} - 12$ $X_{\text{diff}} > 4$	1 px/f

Table 24: 6Leg Values. $X_{\text{diff}} = |X_{\text{6Leg}} - X_{\text{Believed Position}}|$

animation, the Stranger will spawn 3 Glowing Bullets (bouncing ball movement with initial speed 5 and increasing speed by 1 Every 00”-05). When the shooting animation is over, Y Speed is randomly set to either 1 or -1.

Stranger	Shooting Frequency	Shot Directions
4:10 Left Stranger	Every 02”-15	31, 0, 1
4:11 Right Stranger	Every 01”-96	15, 16, 17
4:21 Right Super Stranger	Every 01”-50	14, 16, 18
4:22 Left Super Stranger	Every 01”-00	30, 0, 2

Table 25: Stranger Values

A.3.12 4:x Runners

The Runners have two states, running and stopped/shooting. They will toggle this state on a fixed Every timer. When first loaded with a screen, they will always start in the stopped state, regardless of Every Event cycles. Every time the Runner enters the running state, it will begin a bouncing bullet movement with a random horizontal speed between 8 and 12 inclusive. While the Runner is in the stopped state it will turn to face Juni every frame. After being in the stopped state for a given number of frames, it will shoot. The green runner (4:16 Homing Bullet Runner) fires a homing bullet with initial Y Speed -2. The two fire-shooting runners shoot a fireball. If facing to the right, the fireball will choose a random direction from 3, 4, or 5. If facing left, the direction will be random from 11, 12, or 13. The speed of the fireball is calculated using the following formula:

$$\text{speed} = 20 + \left|\frac{X_{\text{Juni}} - X_{\text{Runner}}}{5}\right| + \left|\frac{Y_{\text{Juni}} - Y_{\text{Runner}}}{4}\right|$$

A.3.13 4:x Spiders

The Spiders use a bouncing ball movement, stopping when they turn around at a wall. Their initial state is stopped when first loaded with a screen, and 4:9 Spider will face towards Juni

Runner	Toggle Move/Stop	Frames before Attacking
4:12 Strange Runner	Every 01”-00	24
4:15 Another Strange Runner	Every 00”-75	18
4:16 Homing Bullet Runner	Every 01”-00	24

Table 26: Runner Values

when first loaded. Every 00”-75, 4:14 3 Leg Spider has a 1/3 chance of beginning to move with speed 30. 4:9 Spider will begin moving with speed 40 when Juni's y-position is strictly between 4 and 8 pixels above that of the spider. When Juni and the spider are standing on the same tile, Juni's y-position is 5 pixels above that of the spider. This means that she can be up to 2 pixels above ground level and trigger the spider. Because Juni can fall and jump at least 4 pixels per frame, it is possible to cross the spider's activation range in both directions without triggering it.

A.3.14 4:18 Static Spiker

Like many other enemies, the static spiker calculates DistanceToPlayer using a taxicab distance formula. When this distance is less than 90 it will become spiky, and when it is more than 120 it will become safe. The static spiker has a similar quirk as the Spiker Muffs, in that it takes 4 animation frames (1̃2 game frames) to become safe to touch after leaving the activation radius, meaning that it is possible to die to it despite having a hologram far away if there are well-placed Shifts or Warps present.

A.3.15 5:1 Scary Shadow Person

The Scary Shadow Person behavior is as simple as it looks. The shadow person will face left when Juni's x-position is to the left, and right when she is to the right. The rest of the behavior uses a variable called Transparency (a slight misnomer, it should really be Opacity, since it controls how visible the shadow person is, not how see-through). Transparency starts at 0. When Juni is facing towards her, she will increase Transparency by 1 every frame until it reaches 128, and decrease by 1 every frame when Juni is facing away until it reaches 0. When Transparency is greater than 120, the shadow person will become deadly, and when it is less than 120 she will become peaceful again. The MMF2 semi-transparency (which also ranges from 0 to 128, with 0 being fully visible) is calculated by taking 128-Transparency.

If Juni is standing exactly at the same x-position as the shadow person, she will continue to face in the same direction, and always decrease transparency. This is the only instance in which it is possible to have her and Juni facing opposite directions, but not becoming deadly.

Due to a quirk in the code, when the transparency reaches 128 (supposedly the maximum), it will begin oscillating between 128 and 129 every frame. This doesn't affect the semi-transparency used in rendering, which is always greater or equal to 0. It does mean, however, that it can take one more frame for the shadow person to become safe to touch when Juni turns away, depending on the state of this cycle. The minimum Transparency does work correctly, and will stay at 0.

A.3.16 6:x Krusers and Labyrinth Spikes

The spikes all have a variable for speed, and a direction. Every frame they will move a number of pixels corresponding to the speed in that direction. If that movement happens to put them inside of a wall, they will back out. That is to say, they will move the same number of pixels in the opposite direction, effectively placing them where they were before. This will also trigger them to randomly change direction (and speed for the Random Labyrinth Spikes). The Krusers will toggle between moving up and moving down. The Labyrinth Spikes will move in a direction orthogonal to their current one (moving up or down will become left or right, and vice-versa).

The code for the Krusers is slightly more simplified than that of the Labyrinth Spikes, which makes them attempt to back out in both directions when inside of a wall (resulting in a net upwards movement). On top of that, because the downwards collision check happens last, they will end up moving up in the regular portion of the movement cycle. This makes them move upwards twice as fast as normal, and produces the fastest upwards speed they can obtain. Conversely, the Labyrinth Spikes will always revert to their previous position after attempted movement in any direction, so placing them inside of a wall will cause them to remain stationary. If they are only slightly embedded in a wall, however, then only the movement away from the wall will successfully push them out, and so they are guaranteed to begin moving in that direction. This can be used in level creation to circumvent the fact that they usually always start by moving up.

Trap	Speed(s)	Initial Speed and Direction
6:1 Kruser	1 (Up) 2 (Down)	1, up
6:2 Labyrint Spike	1	1, up
6:3 Random Labyrinth Spike	{1, 2}	2, up
6:4 Fast Labyrinth Spike	2	2, up

Table 27: Spike Values

The Labyrinth Spikes are all 24 pixels wide by 24 pixels high, meaning that they fit nicely inside of a tile. Provided the rest of the terrain is all perfectly grid-aligned, this means that they too will remain aligned to the grid. It is possible for them to become offset by colliding with a wall that it not exactly at a multiple of 24. It helps that their hitboxes are exactly the same shape as they are, meaning that there are only really 2 pixels[22] colliding with the walls (where the spikes end). If there is only one pixel in one of those two locations, they will continue to move on the grid. If there are 22 pixels in neither of those locations, then the spikes will effectively get stuck in the wall, causing them to perform a 180 degree turn around. Knowledge of the Labyrinth Spike movement can be used to design special labyrinths to make them move in unique ways.

The fact that labyrinth spikes will usually move along a 24 pixel grid can also give the illusion that if a labyrinth spike enters a corner that it will randomly choose one of two directions to exit it. In fact, there is nothing to prevent the labyrinth spike from randomly choosing directions that are into a wall. The movement/collision checks happen in the same

[22] or 8 if they are moving at 2 pixels per frame

order every frame, first testing upwards movement, then left, then down, then right. It is possible for a labyrinth spike to bonk on three different walls and still leave on the same frame if it goes in that order. If it attempts to go in the reverse order (right, then down, then left, then up), it will take three extra frames to finally move up. Theoretically, it is possible for a labyrinth spike to become stuck in a corner forever if it randomly chooses the wrong direction every time. This randomness in the time it takes labyrinth spikes to navigate corners can make otherwise predictable cycles become inconsistent. The full probability tables for labyrinth spike movement can be found in Fig. 26.

A.3.17 6:5 Eater

When Juni touches the eater, it will change animation to eating and become deadly. Usually, this will cause a death, because the eating hitbox is reasonably larger than the teeth hitbox. It is possible to survive for a couple extra frames by jumping on the same frame as activating it, in the middle of the teeth, but the mouth closes too quickly to escape. There are a couple ways to trigger the eating animation and survive under special circumstances. If there is no ground beneath the eater, it is possible to trigger it on one frame and then fall beneath it on the next. With a close enough Shift object, it is possible to warp away on the same frame as triggering it. Finally, it is worth noting that there is exactly enough space between the two center teeth to stand safely, although doing so is pixel perfect.

A.3.18 6:6 Trap Fire

The Trap Fires are stationary, and shoot a fireball on a fixed timer. It's a standard frame countdown, with an initial value of 50, and resetting to 150 every time it hits 0. When the frame counter hits 0, the Trap Fire begins a shooting animation, launching the fireball at the end of that animation. The animation speed is a random value between 18 and 27 inclusive, and it shoots on frame 7, meaning that it can theoretically take anywhere between 25 and 38 frames (or possibly 26 and 39; this has not been precisely tested), and that there are 10 different times at which it may choose to shoot.

Interestingly, only one trap fire is allowed to shoot at a time. This means that multiple ones that choose the same random animation speeds will shoot one by one. If there are sufficiently many of these, then it is possible for them to run out of frames during their attack animation, at which point it will loop and they will have to wait another cycle to get a change to shoot again.

A.3.19 6:x Absurd

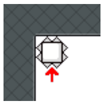
The absurd enemies all behave similarly. When Juni's x-position gets within a certain number of pixels to theirs, they will attack. After attacking, the Absurd Self Dropper will be unable to attack again, and the Absurd Stuff Dropper and Absurd Stuff Shooters will start a countdown to when they are able to attack again. For both of these, this countdown is 200 frames.

When the Absurd Self Dropper attacks, his body becomes deadly, and a vertical speed is set to 1. This speed is incremented by 1 Every 00"-07, possibly including the first frame, until it reaches 9. Every frame, this speed is added to the y-position. Because it uses an Every

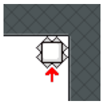
Visual representation of corner and direction of entry

	Direction the spike exits
Frames that the spike gets delayed	Probability of each combination

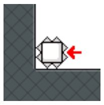
Key



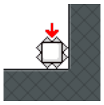
	Right	Down
0	50.0%	25.0%
1	12.5%	6.3%
2	3.1%	1.6%
3	0.8%	0.4%
4	0.2%	0.1%
5	0.0%	0.0%



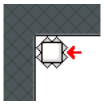
	Left	Down
0	50.0%	
1	12.5%	25.0%
2	3.1%	6.3%
3	0.8%	1.6%
4	0.2%	0.4%
5	0.0%	0.1%



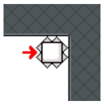
	Up	Right
0		25.0%
1	50.0%	6.3%
2	12.5%	1.6%
3	3.1%	0.4%
4	0.8%	0.1%
5	0.2%	0.0%



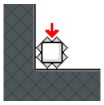
	Left	Up
0		
1	50.0%	25.0%
2	12.5%	6.3%
3	3.1%	1.6%
4	0.8%	0.4%
5	0.2%	0.1%



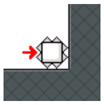
	Down	Right
0	50.0%	
1	12.5%	25.0%
2	3.1%	6.3%
3	0.8%	1.6%
4	0.2%	0.4%
5	0.0%	0.1%



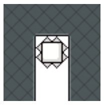
	Down	Left
0		
1	50.0%	25.0%
2	12.5%	6.3%
3	3.1%	1.6%
4	0.8%	0.4%
5	0.2%	0.1%



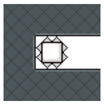
	Right	Up
0	50.0%	
1	12.5%	25.0%
2	3.1%	6.3%
3	0.8%	1.6%
4	0.2%	0.4%
5	0.0%	0.1%



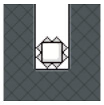
	Up	Left
0		
1	50.0%	25.0%
2	12.5%	6.3%
3	3.1%	1.6%
4	0.8%	0.4%
5	0.2%	0.1%



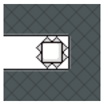
	Down
0	25.0%
1	37.5%
2	18.8%
3	9.4%
4	4.7%
5	2.3%
6	1.2%
7	0.6%
8	0.3%



	Right
0	25.0%
1	37.5%
2	18.8%
3	9.4%
4	4.7%
5	2.3%
6	1.2%
7	0.6%
8	0.3%



	Up
0	
1	25.0%
2	37.5%
3	18.8%
4	9.4%
5	4.7%
6	2.3%
7	1.2%
8	0.6%



	Left
0	
1	25.0%
2	37.5%
3	18.8%
4	9.4%
5	4.7%
6	2.3%
7	1.2%
8	0.6%

Figure 26: Probabilities of a Labyrinth Spike leaving a corner in a specific direction and after a specific number of frames

Event, this introduces a bit of randomness to the movement. When the enemy overlaps the ground, it will push itself back out, stop falling, and become harmless once more.

When the other two attack, they will spawn a projectile on the third animation frame of their attack animation (note that the first frame is identical to their idle animation). This projectile has a constant Y Speed value, which is added to the y-position every frame. It does not move left or right. When the projectile collides with a wall, it will back out one half-step (half of the Y Speed value), and then break apart into 17 smaller projectiles that move outwards in a half circle. These smaller projectiles use bouncing ball movement, with speed 5 and increasing speed by 1 Every 00”-05. Because only the smaller projectiles use an Every Event, the movement of the large shots is consistent, but the 17 small shots may have some randomness.

Absurd	**XDiff attack range (pixels)**	**Projectile Y Speed (pixels per frame)**
6:7 Absurd Stuff Dropper	(-5, +5)	4
6:8 Absurd Self Dropper	(-12, +12)	
6:9 Absurd Stuff Shooter	(-72, +72)	-6

Table 28: Absurd Values

A.3.20 6:x Spikes

The spikes use a simple Distance to player algorithm, expanding when Juni gets close, and contracting when she moves away. Unlike most enemies, however, this Distance to player is calculated using the Euclidean metric, i.e.

$$\text{Distance to player} = \sqrt{(X_{\text{Spikes}} - X_{\text{Believed Position}})^2 + (Y_{\text{Spikes}} - Y_{\text{Believed Position}})^2}$$

The hotspot for the spikes (the position used to calculate the distance) is located in the center of the tile containing the spikes. As soon as the spikes enter the expanding animation, they are deadly, and as soon as the spikes enter the contracting animation, they are safe. This means that it is possible to touch them while contracting and not die, provided a fast enough way to cross the distance (Shifts, screen transitions, hologram, etc.)

Spikes	**Expand distance**	**Contract distance**
6:10 Up Spikes	< 60	> 100
6:11 Down Spikes	< 70	> 110
6:12 Left Spikes	< 55	> 95
6:13 Right Spikes	< 65	> 105

Table 29: Spikes Values

The hitbox of each spike (when fully extended) consists of 4 triangles, each 5 pixels wide and 3 pixels high. Because the four spikes must cover a full tile, each individual spike covers 6 pixels, leaving a 1 pixel gap to one side. This means that each of the spikes has one pixel on the side where it is possible to touch the wall, floor, or ceiling, but not the spikes. For the floor spikes, this pixel is on the left, meaning that it is safe to stand on the far left side of a spike platform. The rest of the spikes are rotated copies of the floor spikes. Thus, for the ceiling spikes the safe pixel is on the right, and so on for the wall spikes. That also means that it is slightly easier to jump over right-facing spikes than left-facing spikes.

A.3.21 8:17 White Explosion

As soon as the white explosion object is fully spawned and active, it will create three "Mega Bomb Explode" fast loops. Each one of these loops will cycle through the 32 bouncing ball directions, spawning a shockwave projectile for each one, with a random speed between 50 and 89, inclusive. Because there are three of these loops, three projectiles will be created in each direction. These shockwave projectiles will then expand outward, creating the explosion effect. The shockwave particles slow down using MMF2's bouncing ball deceleration (with a value of 20). When their speed reaches 0, they disappear.

Due to a bug in the code[23], having more than one white explosion on screen will cause peculiar behavior. Ordinarily, three fast loops are created that spawn 32 shockwave projectiles each, one for each direction. When there is more than one white explosion object created, three fast loops are created for each one, but also each loop will create objects for every single White Explosion on screen. Consequentially, one white explosion object will have three shockwave particles in each of the directions. Two white explosions will each have 6 shockwaves per direction. Three white explosions will have 9 each, and so on. Thus, the number of projectiles in total will increase as the square of the number of white explosion objects on screen.

The next quirk of having multiple white explosions on screen is that MMF2 will only set the speed correctly for the last one created per any given loop cycle. When there is only one white explosion on screen, every shockwave particle has its own loop cycle. When there are more than one, however, the loops will create the objects, starting at the white explosion highest up, and then moving left to right and top to bottom like a typewriter. The vertically lowest white explosion (or horizontally farthest right if there is a tie) is therefore the only one whose projectiles will be the correct speed. All the other white explosion objects will spawn projectiles which move at default speed, which happens to be 30.

There are other MMF2-related glitches, such as some directions occasionally not spawning projectiles (this could be either accidental despawning, directions not being set properly, or occasional spawning failure itself). These minor bugs occur randomly, and don't seem to be reproducible, which suggests that they have to do with the MMF2 framework not being perfectly consistent (such as with the glitches mentioned in section 5.4).

A.3.22 11:x Discs

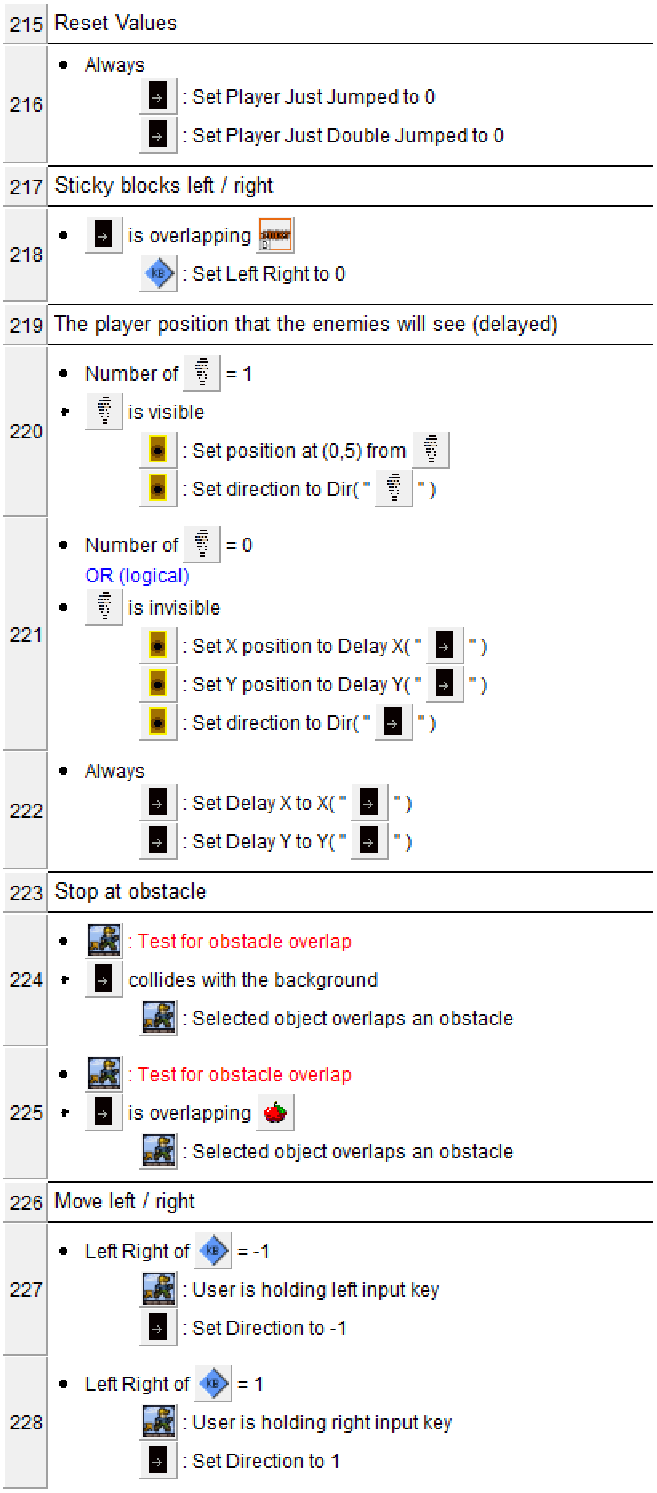
The Discs obey a similar Bouncing Bullet movement as the Mufs, periodically randomizing direction and speed.

[23] This is mostly due to MMF2's weird design of object behaviors, and lack of synergy with fast loops.

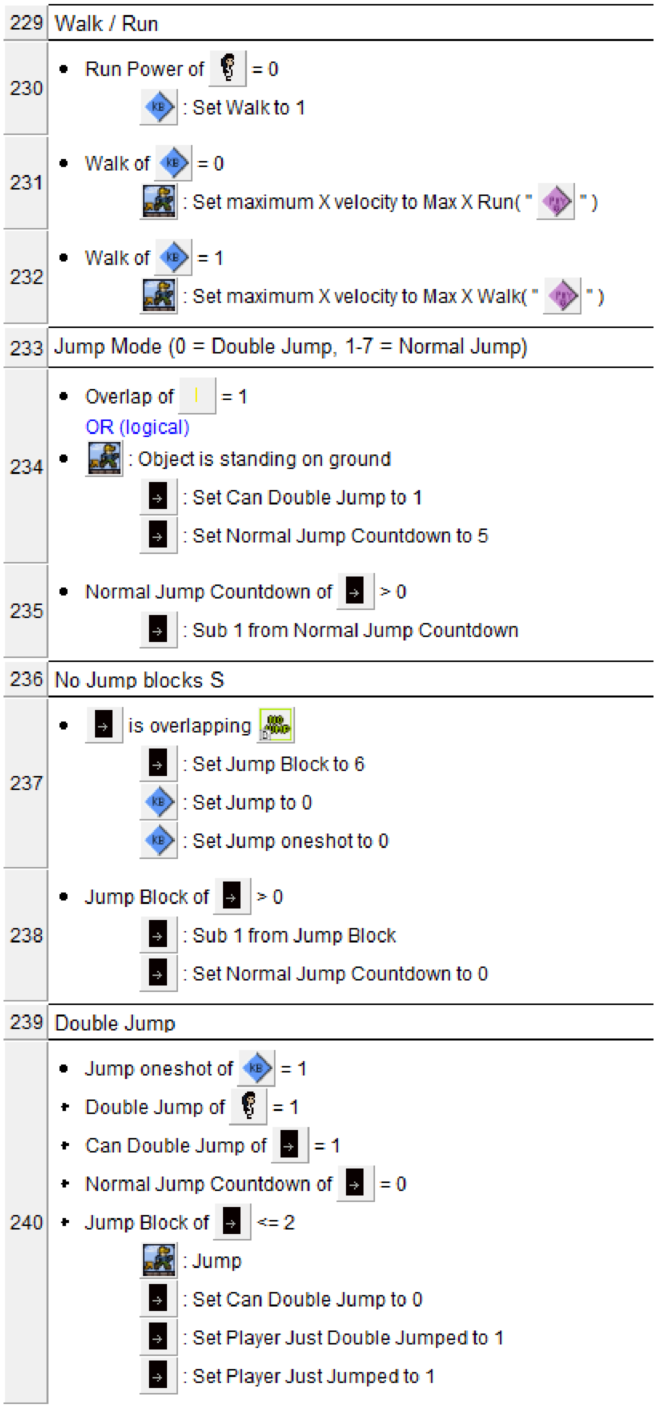
Disc	**Change Direction**	**Change Speed**	**Initial Speed**	**Random Speeds**
11:1 Reddisc	Every 00”-88	Every 00”-30	8	{0,3,6}
11:2 Shockdisc	Never	Every 00”-55	8	{0,3,6}
11:3 PurpleDisc	Every 00”-65	Every 00”-35	8	{0,4,8}
11:4 PurpleDisc w spikes	Every 00”-54	Every 00”-23	8	{0,4,8}
11:5 Reddisc w Spikes	Every 00”-65	Every 00”-35	8	{0,3,6}
11:6 Shockdisc 2	Never	Every 01”-27	8	{5,9}
11:7 Another Disc	Every 01”-65	Every 00”-45	8	{0,4,8,12}
11:8 Greendisc	Every 00”-65	Every 00”-35	8	{0,3,6}
11:9 Shockdisc 3	Never	Every 01”-27	8	{5,9}
11:10 Yellowdisc	Every 00”-77	Every 00”-45	8	{0,9,18}

Table 30: Disc Values

B Movement Code



```
215 Reset Values
216 • Always
          : Set Player Just Jumped to 0
          : Set Player Just Double Jumped to 0
217 Sticky blocks left / right
218 •   is overlapping
          : Set Left Right to 0
219 The player position that the enemies will see (delayed)
220 • Number of   = 1
    •   is visible
          : Set position at (0,5) from
          : Set direction to Dir( "   " )
221 • Number of   = 0
      OR (logical)
    •   is invisible
          : Set X position to Delay X( "   " )
          : Set Y position to Delay Y( "   " )
          : Set direction to Dir( "   " )
222 • Always
          : Set Delay X to X( "   " )
          : Set Delay Y to Y( "   " )
223 Stop at obstacle
224 •   : Test for obstacle overlap
    •   collides with the background
          : Selected object overlaps an obstacle
225 •   : Test for obstacle overlap
    •   is overlapping
          : Selected object overlaps an obstacle
226 Move left / right
227 • Left Right of   = -1
          : User is holding left input key
          : Set Direction to -1
228 • Left Right of   = 1
          : User is holding right input key
          : Set Direction to 1
```



229 Walk / Run

230
- Run Power of = 0
  - : Set Walk to 1

231
- Walk of = 0
  - : Set maximum X velocity to Max X Run(" ")

232
- Walk of = 1
  - : Set maximum X velocity to Max X Walk(" ")

233 Jump Mode (0 = Double Jump, 1-7 = Normal Jump)

234
- Overlap of = 1

  OR (logical)
- : Object is standing on ground
  - : Set Can Double Jump to 1
  - : Set Normal Jump Countdown to 5

235
- Normal Jump Countdown of > 0
  - : Sub 1 from Normal Jump Countdown

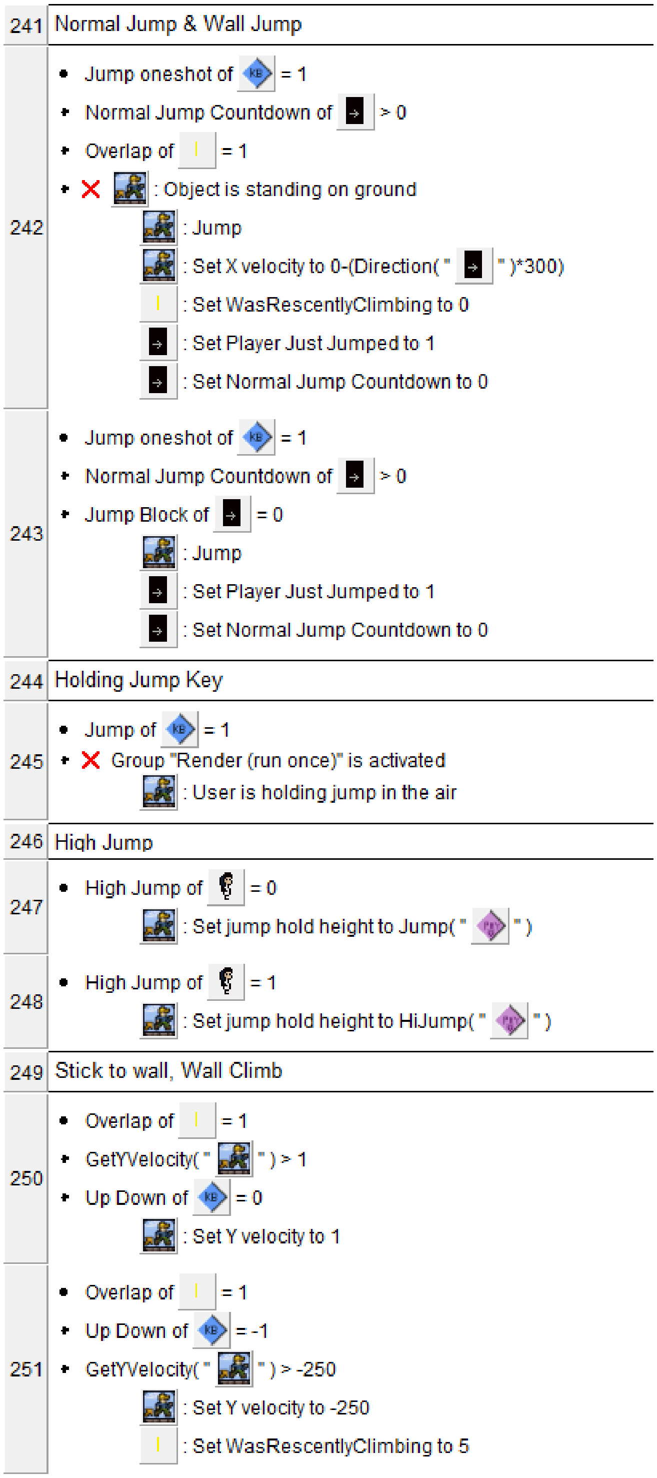
236 No Jump blocks S

237
- is overlapping
  - : Set Jump Block to 6
  - : Set Jump to 0
  - : Set Jump oneshot to 0

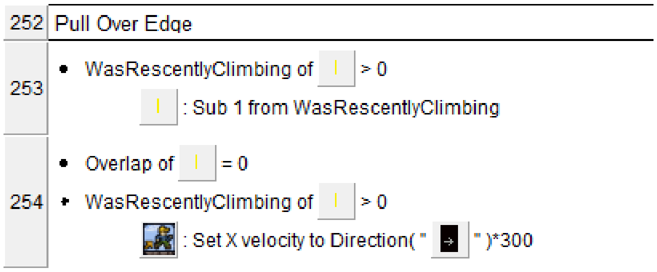
238
- Jump Block of > 0
  - : Sub 1 from Jump Block
  - : Set Normal Jump Countdown to 0

239 Double Jump

240
- Jump oneshot of = 1
- Double Jump of = 1
- Can Double Jump of = 1
- Normal Jump Countdown of = 0
- Jump Block of <= 2
  - : Jump
  - : Set Can Double Jump to 0
  - : Set Player Just Double Jumped to 1
  - : Set Player Just Jumped to 1



```
241  Normal Jump & Wall Jump

242  • Jump oneshot of  = 1
     • Normal Jump Countdown of  > 0
     • Overlap of  = 1
     • ✗  : Object is standing on ground
            : Jump
            : Set X velocity to 0-(Direction( "  " )*300)
            : Set WasRescentlyClimbing to 0
            : Set Player Just Jumped to 1
            : Set Normal Jump Countdown to 0

243  • Jump oneshot of  = 1
     • Normal Jump Countdown of  > 0
     • Jump Block of  = 0
            : Jump
            : Set Player Just Jumped to 1
            : Set Normal Jump Countdown to 0

244  Holding Jump Key

245  • Jump of  = 1
     • ✗ Group "Render (run once)" is activated
            : User is holding jump in the air

246  High Jump

247  • High Jump of  = 0
            : Set jump hold height to Jump( "  " )

248  • High Jump of  = 1
            : Set jump hold height to HiJump( "  " )

249  Stick to wall, Wall Climb

250  • Overlap of  = 1
     • GetYVelocity( "  " ) > 1
     • Up Down of  = 0
            : Set Y velocity to 1

251  • Overlap of  = 1
     • Up Down of  = -1
     • GetYVelocity( "  " ) > -250
            : Set Y velocity to -250
            : Set WasRescentlyClimbing to 5
```



```
252  Pull Over Edge
253  • WasRescentlyClimbing of  > 0
              : Sub 1 from WasRescentlyClimbing
254  • Overlap of  = 0
     • WasRescentlyClimbing of  > 0
              : Set X velocity to Direction( "  " )*300
```

C Python Script for Optimizing Umbrella Pumps

This is a script written in the Python programming language. It takes the measurements of a gap as input and prints out the optimal (or close to it) umbrella pumps to cross. The script is also available to download online at `http://www.ksruns.com/resources/umbrella_pump_calculator.py`.

```
# umbrella_pump_calculator.py
# Python 3.6.2

import math
import sys

console_detail = 3

def console_print(string, detail):
    if (detail <= console_detail):
        print(string)

def input_number(prompt):
    while(True):
        inputStr = input(prompt)
        try:
            return int(inputStr)
        except ValueError:
            print(”Error: Input is not an integer.”)

def input_bool(prompt):
    while(True):
        inputStr = input(prompt + ” [Y/N]”)
        if (inputStr.lower() == ”y” or inputStr.lower() == ”yes”):
            return True
        elif (inputStr.lower() == ”n” or inputStr.lower() == ”no”):
            return False
        else:
            print(”Error: Input should be either Y or N.”)

def array_add_row(arr, width):
    new_row = []
    for x in range(0, width):
        new_row.append(”null”)
    arr.append(new_row)

def jump(regular_frames, max_pumps):
    # The width of the array:
    #   1 column for ledge fall off
    #   1 or 2 columns for jump or double jump
    #   the rest for the umbrella dips
```

```
width = 3 + max_pumps
# Create the empty array
frames = []
# Populate the array with velocities
for i in range(0, regular_frames):
    x = -1
    y = 0
    while True:
        # Scan left-to-right
        x += 1
        # When reaching the end of a row move to the beginning of the next
        if x >= width:
            x = 0
            y += 1
        if y >= len(frames):
            array_add_row(frames, width)
        # Skip cells that already have a velocity
        if frames[y][x] != ”null”:
            continue
        # Column 0: Ledge fall off
        if x == 0:
            if ignore_ledge_fall:
                continue
            if y < 21 or y > 27:
                # Too early to line up with the regular jump values
                # or too late to avoid being a double jump
                continue
            elif y == 21:
                value = 0
                break
            else:
                value = frames[y-1][x] + 23
                break
        # Columns 1 and 2: Jump and double jump
        elif x < 3:
            if x == 2 and not double_jump:
                continue
            elif y == 0:
                value = -477
                break
            else:
                value = frames[y-1][x] + 23
                break
        # Columns 3+: Umbrella dips
        else:
            if y < 26:
                continue
                elif y == 26:
                    # Avoid multiple one-frame umbrella pumps
                    already_a_one_frame_pump = False
                    for u in range(3,x):
                        if (
                            frames[y][u] == 126 and
                            (
                                len(frames) <= y+1 or
                                frames[y+1][u] == ”null”
                            )
                        ):
                            already_a_one_frame_pump = True
                            break
                    if already_a_one_frame_pump:
                        continue
                    else:
                        value = 126
                        break
                else:
                    value = frames[y-1][x] + 23
                    break
        # Terminal velocity
        if value > 700:
            value = 700
        # Add that value to the array
        frames[y][x] = value
        # A 138 + 144 can be replaced with an extra 126 + 149 umbrella pump
        if not ignore_ledge_fall and x == 1 and y == 27:
            for u in range(3,width):
                if frames[26][u] == ”null”:
                    frames[27][0] = ”null”
                    frames[27][1] = ”null”
                    frames[26][u] = 126
                    frames[27][u-1] = 149
                    break
        # If there are no 138's, then we replace 144 + 144 instead
        if ignore_ledge_fall and x == 2 and y == 27:
            for u in range(3,width):
                if frames[26][u] == ”null”:
                    frames[27][1] = ”null”
                    frames[27][2] = ”null”
                    frames[26][u] = 126
                    frames[27][u-1] = 149
                    break
    return frames

def print_array(arr):
    for row in arr:
        for cell in row:
            if cell == "null":
                print(".\t", end='')
            else:
                print(str(cell) + "\t", end='')
        print()

def jump_displacement(frames):
    displacement = 0
    for row in frames:
        for cell in row:
            if cell != "null":
                displacement += cell
    return displacement


##########################################################################################


# Collect information from the user
console_detail = input_number("How detailed do you want the" +
                              "printouts to be? (0, 1, 2, or 3)")
ignore_ledge_fall = input_bool("Start the jump from a wall? " +
                               "(Ignore the first ledge fall off)")
double_jump = input_bool("Does the user have the double jump powerup?")
gap_x = input_number("Enter the horizontal distance to cover (in centipixels)")
gap_y = input_number("Enter the vertical distance to cover (in centipixels)")
print()

# Begin testing options
console_print("Testing for options", 1)
jump_is_possible = False
best_backup = [999999]
# The maximum number of frames we could spend in umbrella is gap_x/260
for umbrella_frames in range(0, int(gap_x / 260)):
    console_print("Testing " + str(umbrella_frames) + " umbrella frames", 2)

    # Calculate preliminary values
    max_pumps = int(umbrella_frames / 2)
        # Integer truncation means that for an odd number of
        # umbrella frames the last pump will be 3-frames long
    console_print("\tUmbrella pumps: " + str(max_pumps), 3)
    umbrella_distance = umbrella_frames * 260
    console_print("\tHorizontal distance travelled in umbrella: "
                  + str(umbrella_distance), 3)
    regular_distance = gap_x - umbrella_distance
```

```
console_print("\tHorizontal distance travelled in freefall: "
              + str(regular_distance), 3)
regular_frames = math.ceil(regular_distance / 350)
    # Integer truncation will make this one frame short
    # of the end, but that's okay because the y-velocity
    # of the last frame is irrelevant
console_print("\tFrames spent in freefall: " + str(regular_frames), 3)

# Create an optimized jump (using PED)
frames = jump(regular_frames, max_pumps)

# Evaluate the horizontal distance covered by the jump
displacement_x = 350 * regular_frames + 260 * umbrella_frames
console_print("\tHorizontal displacement of jump: " + str(displacement_x), 3)
# Evaluate the vertical distance covered by the jump
#   the sum of all the freefall velocities from the array
#   the sum of all the 103 velocity frames from the umbrella frames
#   22*2 for releasing S before each jump
displacement_y = jump_displacement(frames) + 103 * umbrella_frames + 44
console_print("\tVertical displacement of jump: " + str(displacement_y), 3)

# Test if we end up too far down
# +200 for the 2 pixel step-up
if displacement_y <= gap_y + 200:
    # IF not, it's a successful jump
    console_print("\tSuccess", 2)
    print()
    jump_is_possible = True
    break
else:
    # If so, see how long it would take to climb the wall back up.
    console_print("\tFailed", 2)
    climb_y = displacement_y - (gap_y + 200)
    # Accelerated wall climb speed = 454.5 cpx/f
    adjusted_time = regular_frames + umbrella_frames + (climb_y / 454.5)
    console_print("\tTotal frames including the climb back up: " +
                  str(adjusted_time), 3)
    # If this improves the best wall climb time, replace it
    if adjusted_time < best_backup[0]:
        best_backup = [
                adjusted_time,
                frames,
                umbrella_frames,
                displacement_x,
                displacement_y
        ]
```

```
# Verify that we found a jump that works
if not jump_is_possible:
    print("Could not find a number of umbrella pumps sufficient to cross the gap.")
    print("Showing the best option, including a wall climb at the end.")
    frames = best_backup[1]
    umbrella_frames = best_backup[2]
    displacement_x = best_backup[3]
    displacement_y = best_backup[4]

# Print the results
print("Umbrella frames:\t" + str(umbrella_frames))
print("Final displcement:\t" + str(displacement_x) + ", " + str(displacement_y))
print()
print_array(frames)
print()

# Print the sequence of pixels and velocities
if not input_bool("Do you wish to see a table of the positions " +
                  "and velocities attained over the jump sequence?"):
    sys.exit(0)
frame_tracker = input_number("Enter a starting frame number")
last_ground_x = input_number("Enter a starting x position (in pixels)")
last_ground_y = input_number("Enter a starting y position (in pixels)")
print()

width = len(frames[0])
height = len(frames)
dist_x = last_ground_x
dist_y = last_ground_y

print("frame\txvel\tyvel\txpos\typos")
for x in range(0, width):
    if x > 2:
        if frames[26][x] == "null":
            # Beyond the last umbrella pump
            continue
        elif x == width - 1 or frames[26][x + 1] == "null":
            # Last umbrella pump
            pump_size = umbrella_frames - 2*(x - 3)
        else:
            pump_size = 2
        for i in range(0, pump_size):
            print("{:d}\t\t\t{:.2f}\t{:.2f}".format(frame_tracker, dist_x,
                                                    dist_y))
            print("\t260*\t103*")
            dist_x += 2.6
            dist_y += 1.03
            frame_tracker += 1
    for y in range(0, height):
        if x < 2:
            if (
                frames[y][x] != "null" and
                (
                    y == height - 1 or
                    frames[y+1][x] == "null"
                )
            ):
                # Last frame before jump (let go of S)
                frames[y][x] = frames[y][x] + 22
        if frames[y][x] != "null":
            print("{:d}\t\t\t{:.2f}\t{:.2f}".format(frame_tracker, dist_x,
                                                    dist_y))
            print("\t350\t{:d}".format(frames[y][x]))
            dist_x += 3.5
            dist_y += frames[y][x] / 100
            frame_tracker += 1
# The last frame (where y-velocity doesn't matter)
print("{:d}\t\t\t{:.2f}\t{:.2f}".format(frame_tracker, dist_x, dist_y))
```